ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

User-space guided memory management with eBPF

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

ΚΩΝΣΤΑΝΤΙΝΟΣ Χ. ΜΟΡΕΣ

Επιβλέπων: Γεώργιος Γκούμας
Αναπληρωτής Καθηγητής Ε.Μ.Π.

Αθήνα, Οκτώβριος 2024

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

User-space guided memory management with eBPF

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

ΚΩΝΣΤΑΝΤΙΝΟΣ Χ. ΜΟΡΕΣ

Επιβλέπων: Γεώργιος Γκούμας
Αναπληρωτής Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 11η Οκτωβρίου 2024

.....	.....	.....
Ν. Κοζύρης	Δ. Πνευματικάτος	Γ. Γκούμας
Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Αν.Καθηγητής Ε.Μ.Π.

Αθήνα, Οκτώβριος 2024.

.....

ΚΩΝΣΤΑΝΤΙΝΟΣ Χ. ΜΟΡΕΣ
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Copyright © Κωνσταντίνος Χ. Μορές, 2024, Εθνικό Μετσόβιο Πολυτεχνείο. Με επιφύλαξη παντός δικαιώματος. All rights reserved.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ' ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα. Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευτεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

Περίληψη

Οι συνεχώς αυξανόμενες ανάγκες μνήμης των σύγχρονων εφαρμογών ασκούν ολοένα και μεγαλύτερη πίεση στο υποσύστημα διαχείρισης μνήμης των λειτουργικών συστημάτων. Ταυτόχρονα, το κόστος μετάφρασης εικονικών διευθύνσεων αποτελεί σημαντικό παράγοντα καθυστέρησης των προσβάσεων στη μνήμη. Συνεπώς, οι κατασκευαστές σύγχρονων επεξεργαστών εντάσσουν στις αρχιτεκτονικές όλο και πιο σύνθετο, εξειδικευμένο υλικό για την επιτάχυνση της διαδικασίας της μετάφρασης. Μείζον κομμάτι της υλικής υποστήριξης αποτελούν τα Translation Lookaside Buffers (TLBs), υλικές caches που αποθηκεύουν τις πιο πρόσφατες μεταφράσεις διευθύνσεων. Στοχεύοντας στη βελτίωση της απόδοσής τους, το λειτουργικό σύστημα συνεργάζεται με το υλικό για την υλοποίηση μεγάλων σελίδων (huge pages). Οι μεγάλες σελίδες είναι εικονικά και φυσικά συνεχόμενες περιοχές μνήμης, μεγαλύτερες από 4KiB, που μπορούν να μεταφραστούν χρησιμοποιώντας μία μόνο εγγραφή στο TLB. Ωστόσο, τα σύγχρονα λειτουργικά συστήματα φαίνεται να μην επιτυγχάνουν τα αναμενόμενα οφέλη απόδοσης τους. Αυτό οφείλεται εν μέρει στην υιοθέτηση ευκαιριακών και άπληστων πολιτικών κατανομής μεγάλων σελίδων που δεν λαμβάνουν υπόψη τα υποκείμενα κόστη, οδηγώντας σε λάθος αποφάσεις. Από την άλλη πλευρά, αρχιτεκτονικές όπως οι ARMv8-A και RISC-V εισάγουν νέα μεγέθη μεγάλων σελίδων, συγκεκριμένα 64KiB και 32MiB, περιπλέκοντας περαιτέρω τις πολιτικές κατανομής τους.

Στα επόμενα κεφάλαια παρουσιάζουμε την τεχνολογία eBPF και στη συνέχεια δείχνουμε πώς μπορεί να χρησιμοποιηθεί για την υλοποίηση προσαρμοσμένων πολιτικών διαχείρισης μνήμης. Συγκεκριμένα, εμπνευσμένοι από το CBMM [21], υλοποιούμε μέσω του eBPF μια πολιτική διαχείρισης μεγάλων σελίδων βασισμένη σε ένα cost-benefit μοντέλο. Μέσω eBPF προγραμμάτων καθοδηγούμε τον πυρήνα του Linux ώστε να επιλέξει το πιο ωφέλιμο μέγεθος μεγάλης σελίδας για μία περιοχή μνήμης. Τα οφέλη των διαφορετικών μεγεθών μεγάλων σελίδων τα καθορίζει ο χρήστης, ενώ θεωρούμε τα κόστη εγκατάστασής τους ως σταθερές, υπολογισμένες εμπειρικά από την παρακολούθηση του συστήματός μας. Τέλος, αξιολογούμε το σύστημά μας πειραματικά εκτελώντας τρία φορτία εργασίας. Για το καθένα δημιουργούμε ένα προφίλ των περιοχών μνήμης τους offline χρησιμοποιώντας εργαλεία όπως το DAMON [27] και το SPE της ARM αρχιτεκτονικής για TLB miss sampling [2]. Τα αποτελέσματα αναδεικνύουν ότι το ενδιάμεσο μέγεθος μεγάλης σελίδας 64KiB μπορεί να επιφέρει ίδια επίδοση με αυτό των 2MiB, μειώνοντας ταυτόχρονα το fragmentation και το memory bloat του συστήματος.

Λέξεις Κλειδιά: Πυρήνας του Linux, Διαχείριση μνήμης, Εικονική μνήμη, Χώρος χρήστη, Χώρος πυρήνα, Huge pages, eBPF

Abstract

With the ever growing memory needs of modern applications, the memory management subsystem of Operating Systems (OSes) faces increasing pressure. The focus has shifted to the address translation (AT) overhead, a major contributor to memory access latency, and inherent to the decade-old, yet essential, virtual memory abstraction. Thus, CPU vendors provide increasingly complex hardware caches, Translation Lookaside Buffers (TLBs), aiming to speed up this operation. In order to enhance TLB's efficiency, OS and HW cooperatively implement and support huge pages. Huge pages are virtually and physically contiguous memory areas, larger than 4KiB, that can be translated using a single TLB entry. Modern, widely adopted OSes seem to not quite achieve the expected performance benefits of huge pages, partially due to cost-unaware, opportunistic, greedy allocation policies that lead them to pitfalls. On the other hand, architectures such as ARMv8-A and RISC-V introduce new huge page sizes of 64KiB and 32MiB, further complicating huge page allocation policies.

In the following Chapters, we will introduce an existing, revolutionary technology, named eBPF, and demonstrate how it can be used to implement custom policies in the Linux memory subsystem. Inspired by CBMM [21], we develop an eBPF-based system that guides the kernel's decision on which huge page size, if any, to allocate. Our approach determines the most beneficial huge page size based on user-defined benefits and empirically calculated, fixed system costs associated with promoting memory regions to different huge page sizes. Finally, we evaluate our system by running three workloads, profiling their memory regions with DAMON [27] and TLB miss sampling using ARM's SPE [2]. The results highlight the benefits of utilizing the intermediate 64KiB huge page size, when suitable.

Key Words: Linux Kernel, Memory Management, Virtual Memory, Address Translation, Userspace, Kernelspace, Huge Pages, eBPF

Ευχαριστίες

Θα ήθελα να εκφράσω τις θερμές μου ευχαριστίες στον επιβλέποντα καθηγητή μου, κ. Γεώργιο Γκούμα, για την εμπιστοσύνη που μου έδειξε επιτρέποντάς μου να ασχοληθώ με ένα ιδιαίτερα ενδιαφέρον θέμα και για την πολύτιμη καθοδήγησή του καθ'όλη τη διάρκεια τόσο των σπουδών μου όσο και της εκπόνησης της παρούσας εργασίας.

Επίσης, θα ήθελα να ευχαριστήσω ξεχωριστά τον Στράτο Ψωμαδάκη για την αμέριστη βοήθεια και στήριξη που μου προσέφερε σε οποιαδήποτε δυσκολία αντιμετώπισα. Η συνεργασία του υπήρξε καταλυτική για την επιτυχία της διπλωματικής μου εργασίας.

Στη συνέχεια ευχαριστώ ιδιαίτερα τους καθηγητές κ. Νεκτάριο Κοζύρη και κ. Διονύσιο Πνευματικάτο για τις γνώσεις και τα εφόδια που μου προσέφεραν μέσα από τα μαθήματά τους, τα οποία συνέβαλαν δραματικά στη διαμόρφωση της ακαδημαϊκής μου πορείας.

Τέλος, δεν θα μπορούσα να παραλείψω τους φίλους μου και την οικογένειά μου, των οποίων η συναισθηματική στήριξη και ενθάρρυνση ήταν καθοριστική σε κάθε στάδιο αυτής της προσπάθειας.

Εκτενής Ελληνική Περίληψη

Οι συνεχώς αυξανόμενες ανάγκες μνήμης των σύγχρονων εφαρμογών ασκούν ολοένα και μεγαλύτερη πίεση στο υποσύστημα διαχείρισης μνήμης των λειτουργικών συστημάτων. Ταυτόχρονα, το κόστος μετάφρασης εικονικών διευθύνσεων αποτελεί σημαντικό παράγοντα καθυστέρησης των προσβάσεων στη μνήμη. Συνεπώς, οι κατασκευαστές σύγχρονων επεξεργαστών εντάσσουν στις αρχιτεκτονικές όλο και πιο σύνθετο, εξειδικευμένο υλικό για την επιτάχυνση της διαδικασίας της μετάφρασης. Μείζον κομμάτι της υλικής υποστήριξης αποτελούν τα Translation Lookaside Buffers (TLBs), υλικές caches που αποθηκεύουν τις πιο πρόσφατες μεταφράσεις διευθύνσεων. Στοχεύοντας στη βελτίωση της απόδοσής τους, το λειτουργικό σύστημα συνεργάζεται με το υλικό για την υλοποίηση μεγάλων σελίδων (huge pages). Οι μεγάλες σελίδες είναι εικονικά και φυσικά συνεχόμενες περιοχές μνήμης, μεγαλύτερες από 4KiB, που μπορούν να μεταφραστούν χρησιμοποιώντας μία μόνο εγγραφή στο TLB. Έτσι, αυξάνουν το εύρος εικονικών διευθύνσεων διευθύνσεων που μπορούν να μεταφραστούν από το TLB, το οποίο αυξάνει δυνητικά την πιθανότητα ευστοχίας σε αυτό.

Ωστόσο, τα σύγχρονα λειτουργικά συστήματα φαίνεται να μην επιτυγχάνουν τα αναμενόμενα οφέλη απόδοσης των μεγάλων σελίδων. Μάλιστα, διαδεδομένες εφαρμογές προτείνουν την απενεργοποίησή τους καθώς η χρήση τους καταλήγει να είναι περισσότερο επιζήμια παρά κερδοφόρα [23]. Αυτό οφείλεται εν μέρει στην υιοθέτηση ευκαιριακών και άπληστων πολιτικών κατανομής τους που δεν λαμβάνουν υπόψη τα υποκείμενα κόστη, οδηγώντας σε λάθος αποφάσεις. Από την άλλη πλευρά, αρχιτεκτονικές όπως οι ARMv8-A και RISC-V εισάγουν νέα μεγέθη μεγάλων σελίδων, συγκεκριμένα 64KiB και 32MiB, περιπλέκοντας περαιτέρω την υλοποίηση αποδοτικών, γενικευμένων πολιτικών διαχείρισης μεγάλων σελίδων από τα λειτουργικά συστήματα.

Στα επόμενα κεφάλαια παρουσιάζουμε την υπάρχουσα τεχνολογία eBPF και στη συνέχεια δείχνουμε πώς μπορεί να χρησιμοποιηθεί για την υλοποίηση προσαρμοσμένων πολιτικών διαχείρισης μνήμης. Συγκεκριμένα, εμπνευσμένοι από το CBMM [21], υλοποιούμε μέσω του eBPF μια πολιτική διαχείρισης μεγάλων σελίδων βασισμένη σε ένα cost-benefit μοντέλο. Μέσω eBPF προγραμμάτων καθοδηγούμε τον πυρήνα του Linux ώστε να επιλέξει το πιο ωφέλιμο μέγεθος μεγάλης σελίδας για μία περιοχή μνήμης. Τα οφέλη των διαφορετικών μεγεθών μεγάλων σελίδων τα καθορίζει ο χρήστης, ενώ θεωρούμε τα κόστη εγκατάστασής τους ως σταθερές, υπολογισμένες εμπειρικά από την παρακολούθηση του συστήματός μας.

Τέλος, αξιολογούμε το σύστημά μας πειραματικά εκτελώντας τρία φορτία ερ-

γασίας. Για το καθένα δημιουργούμε ένα προφίλ των περιοχών μνήμης τους offline χρησιμοποιώντας εργαλεία όπως το DAMON [27] και το SPE της ARM αρχιτεκτονικής για TLB miss sampling [2]. Συγκεκριμένα χρησιμοποιούμε τις πληροφορίες που συλλέγονται ώστε να κατατάξουμε τις περιοχές μνήμης σε ορισμένες κατηγορίες [20], οι οποίες δηλώνουν πιο μέγεθος σελίδας είναι το πιο κερδοφόρο.

Τα αποτελέσματα αναδεικνύουν ότι το ενδιάμεσο μέγεθος μεγάλης σελίδας 64KiB μπορεί να επιφέρει ίδια επίδοση με αυτό των 2MiB σε ορισμένες περιπτώσεις. Η αναγνώριση των περιπτώσεων αυτών και η χρήση σελίδων 64KiB είναι καίρια, καθώς τότε μειώνεται ταυτόχρονα το fragmentation και το memory bloat του συστήματος, ενώ η επίδοση του είναι παρόμοια.

Contents

Chapter 1

Introduction

In modern computing, particularly in data centers, the memory management subsystem of operating systems (OS) is under increasing pressure. With workloads growing ever more data-intensive, especially with the rise of machine learning and artificial intelligence (AI) applications, while the chasm between processor and memory speed continuously expands, memory access has become a significant performance bottleneck [34].

The core virtual memory abstraction, conceived decades ago and fundamentally unchanged, is put through the test of time, since application needs are rapidly evolving. The main problem lies within the inherent virtual-to-physical address translation (AT) step to virtual memory, which contributes substantially to memory access latency [7].

As virtual memory proved far too essential to abandon, the industry responded with increasingly complex hardware aimed specifically to alleviate its overheads. Such hardware include various types of multilevel Translation Look-aside Buffers (TLBs), hardware Page Table Walkers (PTWs), specific Memory Management Unit's (MMU) caches, nested TLBs and more.

The performance of TLBs turned out crucial for modern applications, because they effectively reduce the AT step into a cache hit. Targeting to improve their hit ratio by extending their reach OS and HW cooperatively implement and support huge pages. Huge pages are virtually and physically contiguous memory areas, larger than 4KiB, that can be translated using a single TLB entry.

1.1 Research Problem

Despite the promising performance benefits of huge pages, they seem to underperform in modern OSes [10] leading to many applications to advise disabling them [23]. This can be partially attributed to the cost-obliviousness of many MM huge-page policies.

For example, Linux Transparent Huge Pages (THP) [33] try to greedily allocate

a huge page every time a memory area is first touched, not considering that this area might be underutilized or that it does not benefit from being backed by huge pages.

However, setting up a huge page is not cost-free. The kernel first needs to find an available, properly aligned physical memory region. This process is inexpensive if there is an abundance of physical memory, but on a fragmented machine this could trigger costly operations, like compaction. The huge page must then be prepared. In the case of file-backed memory this means fetching the data from the disk, and in the case of anonymous mappings it means zeroing the contents of the huge page.

Therefore, if the costs of setting up a huge page outweigh its benefits not only the current process takes longer, but the whole system is susceptible to more slowdowns due to fragmentation [21]. Consequently, the kernel should take great care deciding whether or not to use huge pages to back application memory.

Several systems have been proposed with more dynamic huge page management policies. Ingens [18] and Hawkeye [25] monitor the system to identify whether or not to promote a memory region to a huge page. They also present new strategies aimed to minimize latency of huge page management. CBMM [21] introduces a more general cost-benefit approach regarding memory policies, including huge pages.

1.2 Motivation

The most common sizes when referring to a huge page used to be 2MiB (the one Linux supported transparently for anonymous memory mappings) and 1GiB. This was due to the fact that popular architectures (e.g x86) provided support for them.

ARMv8-A and RISC-V extend the supported huge page sizes, using a similar mechanism which utilizes unused bits of the page table entries to designate contiguous groups of pages [2, 15, 32, 26].

This futher complicates the OS huge-page policies, by adding 64KiB and 32MiB huge pages to the mix. In order to support these new huge page sizes, Linux switched to a mutli-sized THP mechanism [29, 30]. However, mTHP doesn't allow for fine-grained control over which applications or which parts of the address space of an application should be backed by which size.

Previous work has showed that applications exhibit different needs regarding huge page management. Thus, we argue that the OS memory manager needs to be more flexible in order to navigate the trade-offs of using different huge page sizes for different applications.

1.3 Our proposal

In this thesis, we leverage eBPF to implement custom memory management policies that dynamically adjust the kernel's decision-making process for huge page promotion.

eBPF (extended Berkeley Packet Filter) is a powerful and flexible technology that allows user-defined programs to be executed safely in kernel space. Originally developed for network packet filtering, eBPF has evolved into a general-purpose framework that enables developers to attach programs to various points in the kernel, known as hooks, to perform custom actions when specific events occur.

We implement hooks for eBPF programs in key points in the page fault path, where the kernel decides what page size will be allocated for the faulting memory region. Then, we develop and attach an eBPF program responsible for guiding the kernel to choose the most beneficial page size. To achieve that, we provide our eBPF program with profiling information about the application's memory regions. Specifically, we estimate the benefit of different page sizes on different memory areas through monitoring the memory access frequency and the TLB misses on those areas.

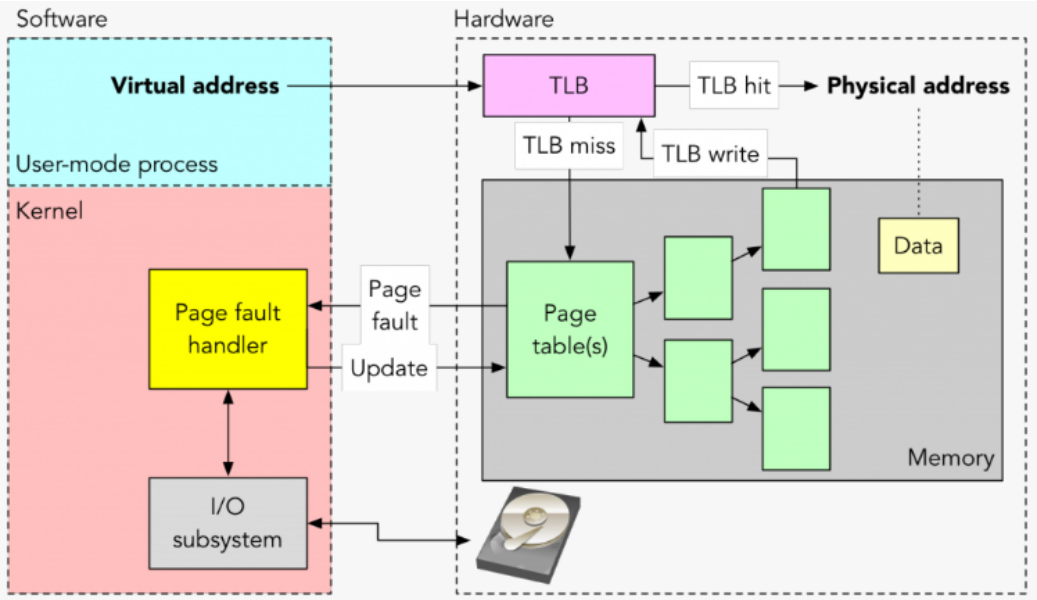
We show that this approach not only improves memory management efficiency but also allows for more flexible and application-specific policies.

Chapter 2

Virtual Memory

This chapter focuses on virtual memory, a core part of the memory management subsystem of modern operating systems. We provide background on virtual memory as a physical memory abstraction and the reasons it dominated the way operating systems handle memory. We continue delving deeper into virtual memory by discussing paging as its most usual implementation. Then, we examine the Address Translation (AT) step inherent in virtual memory and the overhead it introduces. Finally, we explore current mitigations techniques aimed to alleviate the AT overhead like hardware caches, Translation Lookaside Buffers and the use of Huge Pages.

Figure 2.1: Virtual Memory Implementation



The majority of modern, widely adopted operating systems use the virtual memory technique to efficiently manage memory. Virtual memory provides an

abstraction of physical memory. It introduces an interface to programs, the virtual address space, through which they need to interact in order to access physical memory.

With virtual memory, programs perceive the system's memory as a large, contiguous, private block of memory. It alleviates programmers from the burden of manually managing data transfers between main memory and secondary storage, allowing easier execution of applications whose memory needs expand beyond those of physical main memory.

The virtual memory abstraction aided the operating system's support for process multitasking, protection enforcement (process isolation) and data sharing (shared memory). The downside of virtual memory is the requirement of a mechanism to map virtual addresses to physical ones. This virtual-to-physical address translation step is a significant overhead introduced to each memory access. To alleviate this overhead the system's software and hardware cooperate, employing caching-like techniques, hardware assisted translations and the use of larger pages.

2.1 Core Overview

In this section we present a high level overview of the virtual memory abstraction. We present core concepts that most modern operating systems adopt to implement it.

2.1.1 Physical Address Space

We connect the term physical address space to the set of addresses that reference the main memory of the system. When referring to a physical address we mean the location of the data inside main memory (e.g. RAM).

2.1.2 Virtual Address Space

The virtual address space consists of the range of addresses that programs use in their instructions to access memory (e.g. load/store operations). The virtual address space is private to each process and typically much larger than the system's actual main memory. For example, the x86-64 architecture provides 48-bit long virtual addresses, thus allowing the programmer to address 256 TB of memory. These virtual addresses may not indicate where the data is actually stored, but they address the data of a specific process.

2.1.3 Address Translation

Programs use virtual addresses when referring to memory, that must be subsequently translated to physical addresses to access real data in main memory. When a process tries to access data using a virtual address, through load and

store instructions the operating system collaborates with underlying hardware to translate this virtual address to a physical address. Then, the CPU uses the provided physical address to fetch the corresponding data. Also, if the requested data are not present in main memory, so such a translation can not exist, the operating system is responsible to find the data in the secondary storage, bring it to main memory and create the corresponding virtual-to-physical address mapping for that process.

2.1.4 Paging

Paging is a core part of the implementation of virtual memory. Keeping track of every individual byte of a process' address space would be highly inefficient, thus the operating system manages a process' address space in fix-sized data chunks, which are called pages. Process access memory through chunks of virtual addresses (virtual page), which are subsequently mapped to an equally sized physical page frame. The size of a page is architecture dependent and in most architectures the base page size is 4 KB. Modern architectures also support larger pages, for example x86 2MiB and 1GiB, and arm64 and RISC-V additional 64KB and 32MB, the usage of which is explained in a later section.

2.1.5 Page Tables

The operating system controls every process's virtual-to-physical page mapping by managing a lookup table per-process, known as the page table. Each page table entry holds the physical page frame translation of the corresponding virtual page.

Multi-level (hierarchical) Page Tables

Implementing the page table as a single one-dimensional array, even when the virtual address space is managed in pages of 4KiB, results in a gigantic array per process. This is neither feasible, especially in a Symmetric Multiprocessing (SMP) context where multiple processes are executing requiring their Page Tables to be in memory, nor necessary since processes use sparsely their virtual address space, requiring only parts of their page table.

Therefore, page tables are implemented as a multi-leveled, hierarchical, tree-like structure. The root of the tree is called the page table base. It is the only memory location needed to access the whole data structure and it is required to reside in main memory. Intermediate nodes' values contain the memory address of nodes further down the tree, while leaf nodes, frequently called as Page Table Entries (PTEs) contain the needed translation. This approach allows page tables to be allocated on demand, when a process needs to map a region of its virtual address space to physical memory (memory allocation), and not all at once.

Each page table entry (PTE) not only holds the physical page number but also includes metadata such as permission bits (read, write, execute), a valid/invalid flag to indicate whether the entry is in use, and a dirty bit to track modified pages.

Accessing the Page Table

Depending on the size of the base page, virtual and physical addresses are split in two parts; the page offset and the virtual/physical page number.

The first part is the page offset which determines the position of referenced the data inside the corresponding page. The page offset is most commonly defined as the minimum amount of Least Significant Bits required to address an entire base page. For example, with 4KiB base pages, the 12 LSBs are used to define the page offset, assuming byte addressable memory.

The remaining bits of the virtual or physical address constitute the number of the virtual/physical page. The virtual page number of virtual addresses is used as index for accessing the page table, in order to retrieve the physical page frame it corresponds to. To achieve this in the context of multi-level, hierarchical page tables, the virtual page number is further broken down in more parts depending on the structure of the hierarchy. Specifically, it is divided in the same number of parts as the number of page table levels. Each part is assigned the minimum sufficient number of bits to index the corresponding level of the page table.

When a memory instruction requires an address translation, the Page Table hierarchy is traversed to acquire it, a procedure called page table walk. Page table walk latency is the main source of overhead in Virtual Memory. Each access to a level of the Page Table hierarchy equals to an independent memory access. Thus, a single load/store operation requires an additional N (in systems with Page Tables of N levels) memory accesses.

Also, this overhead exhibits quadratic growth in virtualized execution. In the most virtual machine implementations each memory access on the guest requires the execution of page table walk on the host's Page Table hierarchy.

This overhead greatly degrades execution time.

Page Table Hierarchy Structure

The structure of the Page Table hierarchy is dictated by one major decision in the system's architecture; whether the Page Table Walk is managed by specialized underlying hardware (the MMU) or software (the Operating System).

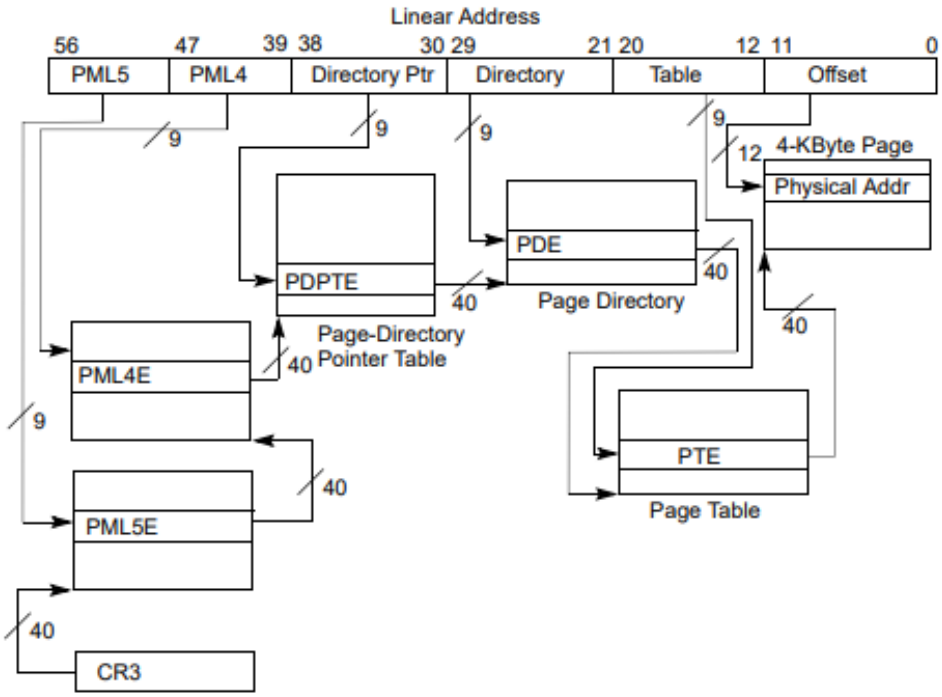
Most modern systems sacrifice the flexibility of the Operating System to decide the organization of the Page Table hierarchy by offloading the Page Table Walk operation to the MMU. This way, Page Table Walks are done transparently to the system's software without the need of executing extra instructions. This approach, enables other independent instructions to continue, while the instruction triggering the Page Table Walk just stalls. Avoiding extra instructions also

ensures performance stability on the system, while there is no interference with the current state of the data and instruction caches.

The Linux operating system supports up to 5 levels of page tables. These levels are named from the highest to the lowest as: PGD (Page Global Directory), P4D (Page 4th Directory), PUD (Page Upper Directory), PMD (Page Middle Directory) and Page Table (PT).

Architecture specific code leverages this interface in order to describe the Page Table hierarchy the Memory Management Unit supports. In the following Figure we provide Intel's 5-level Page Table organization, an extension of the original 4-level Page Table enabled by setting the CR4 register.

Figure 2.2: Intel's 5-level Page Table Hierarchy



2.2 Address Translation Hardware

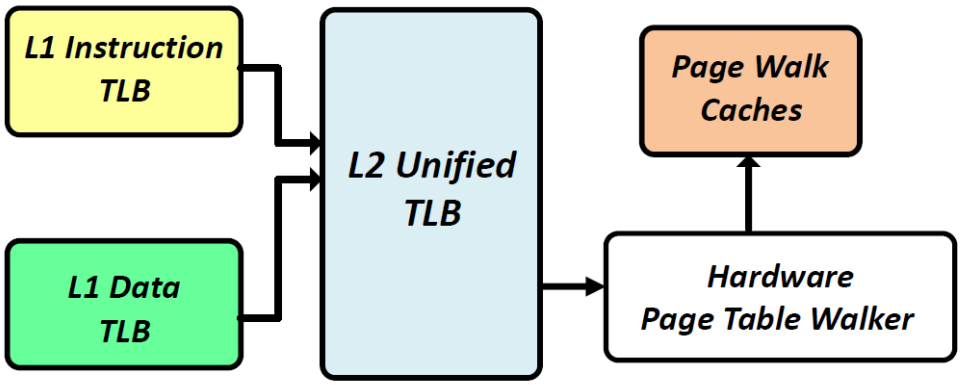
As we mentioned in the previous section, the Memory Management Unit (MMU) of modern CPUs is the backbone of Virtual Memory. It employs specialized hardware components such as caches (Translation Lookaside Buffers, Page Table Walk Caches) and finite state machines (Page Table Walkers) to ensure that the address translation overhead is minimal. The operating system is responsible to maintain and update the information on the Page Table hierarchy, which is sub-sequentially consumed by the MMU. Also, it is responsible for managing effi-

ciently the main memory of the system, fetching to it data required for processes to execute.

Following we describe the three major hardware components of the MMU:

- Translation Lookaside Buffer (TLB)
- Page Table Walker
- Page Table Walk Caches

Figure 2.3: Intel's Skylake MMU design



2.2.1 Translation Lookaside Buffers

The most important hardware component of the MMU is the Translation Lookaside Buffer (TLB), a cache which stores recent virtual-to-physical address mappings. In many cases an under-performing TLB (low hit-ratio) is the culprit for major application slowdowns.

Usage

When a load/store instruction references a virtual address, the TLB is first checked for the corresponding translation. If the TLB has the translation stored the corresponding physical address is provided to the CPU. This way, it attempts to minimize the address translation overhead by avoiding expensive page table walks for frequently accessed memory addresses.

When a requested address translation is not found in the TLB, a TLB miss occurs. The instruction that triggered the TLB miss stalls, while the MMU executes a page walk to provide the requested translation. If the page is not present in memory, a page fault is triggered, prompting the OS to load the required page from disk.

The amount of memory that is accessible only by the TLB is called the TLB reach. The TLB reach is a critical factor in the performance of an application. If the application's working set doesn't fit in the TLB, thrashing of the TLB may occur causing multiple page table walks, costing a lot of CPU cycles and harming performance. For example, if a computer system uses only base pages of 4KB and a CPU core has 16 entries, then the TLB reach is 64 KB. It is observed that the TLB reach can be expanded by increasing the size of the pages used to manage virtual memory.

Design

The TLB follows the usual cache design choices, regarding placement, replacement policies and structure.

For example, most TLBs are indexed using the lower bits of the virtual page number. The tag is comprised by the unused bits of the virtual page number plus the process's ID, or more specifically an Address Space Identifier (ASID). This helps recognizing which translation corresponds to which tasks, avoiding unnecessary TLB shoot-downs on context switches.

A TLB can be direct-mapped, n-way set associative or fully associative. Furthermore, the most common replacement policy when the TLB entries are full is the Least Recently Used (LRU) algorithm.

Finally, most TLBs are organized, similar to data caches, in hierarchic faster-to-slower entries. An address translation lookup is first tried on a small L1 TLB cache. If it misses, the next, normally larger levels of the TLB hierarchy are tried (usually one extra level). If the translation is not found in any level of the TLB hierarchy-i.e the lookup misses in the last level of the TLB hierarchy, a Page Table Walk is bound to be executed.

2.2.2 Page Table Walkers

The Page Table Walker (PTW) implements in hardware the traversal of the Page Table tree and retrieves the physical translation of a virtual page. This avoids the need of expensive context switches and software handling. Also, implementing Page Table Walks in hardware, allows the pipelining of TLB misses and enables concurrent TLB misses.

The PTW consists of two main components:

- A state machine designed based on the architecture's specific page table structure.
- Registers able to keep track of outstanding TLB misses.

Since page tables are unique to each process, the CPU keeps track of the executing process' page tables through a special register called page table page register (i.e CR3 in x86). This register contains the physical address of the root of

the page table of the executing process. It is the Operating System's job to update this register when a process is scheduled into a CPU core (context switch). This register is subsequently used from the PTW's state machine.

2.2.3 Page Table Walk Caches

Along with Page Table Walkers the MMU employs another set of specialized caches storing information about the Page Table hierarchy, named Page Table Walk Caches.

Specifically, they cache intermediary non-leaf levels of the Page Table hierarchy.

This aims to accelerate page table walks, by substituting a high-latency access to the cache/memory hierarchy of the system with a lower latency cache access for acquiring Page Table values.

2.3 Huge Pages

Huge pages, as the name suggests are larger than the base page size (usually 4KiB) pages. They are large contiguous blocks of physical memory and are used to map suitable virtual address regions. For example a virtual memory region of 4MB could be mapped with 1024 pages of 4KiB, which may not be contiguous in physical memory, or with 2 pages of 2MB.

Huge pages do not replace base pages in the entirety of the system. Instead the different page sizes are used in combination, with the operating system deciding which page size to use to back a particular memory region. They aim to reduce the address translation overhead by improving the performance of the TLB component of the MMU.

In order to make huge pages available to a system, the operating system and the underlying hardware must once again cooperate. Together they are responsible for making the Page Table hierarchy aware of the different page sizes, allocating contiguous memory for a huge page (operating system's side) and providing a huge page aware TLB (hardware architecture's side).

2.3.1 Benefits of Huge Pages

Huge pages are the state-of-practice technique to reduce the overheads of address translation. By using huge pages the TLB reach can be significantly increased and also the page walk latency can be reduced, as it will be later explained. Their performance benefits stem primarily from two things:

- Effectively increasing the TLB reach.
- Reducing the length of the Page Table Walk.

TLB reach

Huge pages can significantly increase the TLB's reach. On system's that use only base pages a TLB entry is able to cache the translation of a single base page. Since huge pages are guaranteed to be contiguous in physical memory, the translation of a single TLB entry applies to every address contained in the huge page region, which is typically a multiple of the base page size.

Let's take for example 2MiB huge pages and a TLB with N entries. Using base pages of 4KiB the TLB is able to translate 4N KiB of address ranges without any conflict or capacity misses. If all entries of the same TLB hold 2MiB huge page translations, the same TLB is able to translate 2N MiB of address ranges. The TLB's reach is multiplied by a factor of 512.

This way the working set size of applications is more probable to fit inside the TLB, increasing the potential TLB hit ratio, and subsequently its performance in case it is memory intensive.

Page Walk Length

Certain huge page sizes are able to decrease the Page Walk length, resulting in reduced latency of TLB misses. This happens because some huge page sizes are able to be translated to a huge physical page using intermediate nodes of the Page Table hierarchy.

We present an example. Each entry of the final level of the Page Table hierarchy (Page Table level), translates contains the translation for a base page. Let's assume that the Page Table level contains 512 Page Table entries (PTEs). Each entry of Page Table level above the Page Table, the Page Middle Directory, contains a pointer to the start address of the corresponding Page Table. In this scenario, a huge page with size of 512 times larger than the base page size is able to be translated from a single PMD entry. Subsequently the memory accesses required to traverse the Page Table hierarchy for this address translation are, in this case, reduced by one.

The performance benefits of this becomes even more evident in virtualized execution where nested Page Table hierarchy is used.

2.3.2 Drawbacks of Huge Pages

Few solutions come without trade-offs and huge pages are not the exception. Although huge pages can provide a useful tool to alleviate the downsides of address translation they must be handled with caution, since there are pitfalls the operating system could walk into.

By hastily backing up virtual memory regions with huge pages, we might lead our system to heavy memory fragmentation, or we could starve other processes of memory. They can be the source of heavy internal fragmentation and the waste of precious memory resources. For example, we could be misled by the large size of a

virtual memory region with identical properties and decide to back it with 2MiB huge pages. This bounds us to allocate a 2MiB huge page even when only as little as 1KiB of memory is actually needed.

Setting up huge pages may also introduce heavy latency to applications using them, if their setup costs outweigh their expected benefit. The setup costs consist of the creation of a large enough contiguous block of memory by running defragmentation (a usual operation in long running systems, where fragmentation persists) and the preparation of it. For anonymous memory this means zeroing the entirety of the physical huge page, while for file backed memory it means fetching the data from the disk. If not necessary, this operation does not only degrade performance, but is also a waste of bandwidth and energy.

2.3.3 Architectural Huge Page support

As already mentioned, in most modern computing systems the software and the underlying hardware need to cooperate to provide certain functionalities. Implementing the logic behind huge page memory allocations would provide no benefits for the system, if the underlying hardware was not huge page aware. In the next paragraphs we describe how two of the most widespread architectures, x86 and ARM64, implement huge pages, the differences between them and the various huge page sizes they support.

x86

The x86 architecture, apart from 4KiB base pages, provides architectural support only for huge page sizes that correspond to intermediate levels of the Page Table hierarchy. It achieves that by introducing a bit, which can be either set or cleared, that signifies whether or not the particular Page Table level corresponds to a huge page.

Specifically, x86 enables huge pages in the Page Middle Directory (PMD) and Page Upper Directory (PUD) levels. Since it defines that a PMD entry contains 512 Page Table Entries (PTEs) and a PUD entry contains 512 PMD entries, the supported huge page sizes are 2MiB and 1GiB accordingly.

ARM64

The ARM64 architecture also provides support for huge pages of 2MiB and 1GiB with the same logic as the x86 architecture.

Furthermore, ARM64 enables intermediate huge page sizes, between 4KiB - 2MiB and 2MiB - 1GiB, by adding another bit, that can be either set or cleared, in Page Table entries and Page Middle Directory entries. This bit, named the contiguous bit, represents that the PTE or PMD entry is one of a set of entries that translate to physical 4KiB or 2MiB pages that are contiguous in main memory.

Using as base size 4KiB pages the set contains 16 entries, enabling intermediate huge page sizes of 64KiB and 32 MiB accordingly.

Huge Page Aware TLB

The major benefit of huge pages is the increase of the TLB reach, subsequently leading to fewer TLB misses. That poses a new challenge for TLB architecture, since now it must be aware of the size of the translation it holds, otherwise it risks memory corruption. Following we introduce two ways modern TLBs recognize huge page sizes:

1. The Memory Management Unit provides a separate TLB for each of the huge page sizes. The TLBs are then accessed in parallel and the one with the matching translation, if any, provides it to the CPU.

2. The second approach is using a single TLB, but probing it for the address translation in more than one steps. First, each TLB entry holds information about how many of the bits of the virtual address they store are used as an index, thus signifying the page offset and the size of the huge page. Then, the TLB is accessed in as many steps as huge page sizes, using different subsets of the virtual page number as index until there is a match, or they exhaust all possible huge page sizes.

2.4 Linux Huge Page support

In this section we explore how Linux provides huge pages to processes. Linux Huge Page support is built on top of multiple page size support that is provided by most modern architectures. The two primary ways Linux manages huge pages are through Transparent Huge Pages (THP) [33, 24] and HugeTLB Pages [14]. The first option attempts to allocate huge pages transparently to the user. The second one requires their manually reservation and the proper reconfiguration of the application to use them.

2.4.1 Transparent Huge Pages

THP was introduced in Linux to automate the allocation and management of huge pages, without user intervention. It enables applications to take advantage of huge page with no modifications needed. THP has two main modes of operation:

- Synchronous huge page allocation at page fault time

- Asynchronous huge page promotion through the khugepaged background thread.

Synchronous

In the synchronous mode, huge pages are allocated at page fault time. When a process accesses a memory region that is not yet backed by physical memory, the kernel attempts to allocate a huge page instead of a 4KiB base page (if the memory region is suitable).

If the attempt fails (and the kernel is configured with the DEFRAG option enabled), Linux will invoke de-fragmentation algorithms, in an attempt to create enough space for the huge page. The de-fragmentation process involves memory compaction (moving around pages in main memory to create a sufficient contiguous free region) and memory reclamation (swapping out memory to disk to increase total free memory), which can be a very costly operation.

Asynchronous

To complement the synchronous allocation, Linux also uses an asynchronous method for huge page promotion through the khugepaged daemon.

Khugepaged runs in the background, periodically scanning the memory of running processes to find areas that can be collapsed into PMD-sized huge pages. To do that, it maintains a list of virtual memory areas that are suitable for promotion.

Linux allows users to set limits on the frequency of khugepaged scans. In a limitless scenario, khugepaged is woken up every time a page fault encounters a memory region that is suitable to be backed by PMD-sized huge pages. The virtual memory region is added to the tail of the list and the khugepaged kernel thread is woken up.

mTHP support in THP

Linux introduces the ability to allocate memory in blocks that are bigger than a base page but smaller than traditional PMD-size (as described above), in increments of a power-of-2 number of pages. mTHP can back anonymous memory (for example 16K, 32K, 64K, etc). These THPs continue to be PTE-mapped, but in many cases can still provide similar benefits to those previously described, while latency spikes are much less prominent because the size of each page isn't as huge as the PMD-sized variant and there is less memory to clear in each page fault. As widespread architectures do not support most of these sizes, the benefits of mTHP do not (generally) stem from address translation. In that case, the benefits of mTHP lie in reducing further page faults on the number of individual 4KiB pages covered by the corresponding intermediate size.

Since the asynchronous THP system supports only the promotion to PMD-sized huge pages, transparent support for intermediate huge page sizes (mTHP) is currently only implemented synchronously at page fault time.

To understand how Linux achieves that we need to delve a little deeper in the page fault code path, regarding page allocation. For the sake of simplicity let's

consider that every huge page size supported by Linux is available through the system's configuration.

We present the steps of huge page allocations. The Linux kernel only proceeds to the next step if for some reason the previous one fails.

1. Try to back the faulting memory region with PUD-sized pages (not supported for anonymous mappings, only through hugetlb [14])

2. Try to back the faulting memory region with PMD-sized pages.

3. Iterate through every intermediate page size starting from the largest and try to back the faulting memory region with it (mTHP support).

4. Allocate a base page for the faulting memory region.

It is evident that Linux follows a greedy policy in deciding the most suitable page size to back a virtual memory region. In the most general case, Linux is going to allocate a PMD-sized page for a virtual memory region that meets the address alignment and size criteria for it. If for some reason this allocation fails, Linux employs memory de-fragmentation algorithms to make enough space. Even if enough space is not available at that moment, khugepaged will eventually scan and that region to PMD-sized pages.

2.4.2 HugeTLB Pages

HugeTLB pages [14] are another method supported by Linux to manage large memory pages. Unlike THP, HugeTLB pages reserves a pool of persistent huge pages at kernel startup. The administrator can allocate those on the kernel boot command line by specifying the "hugepages=N" parameter, where 'N' = the number of huge pages requested.

Once a number of huge pages have been pre-allocated to the kernel huge page pool, a user with appropriate privilege can use either the mmap system call or shared memory system calls to use the huge pages.

This method is often used in performance-critical applications where developers need precise control over memory allocation.

It is important to mention that the HugeTLB interface is the only way for a user to allocate PUD-sized (commonly 1GiB) huge pages for a memory region. While the transparent mode supports both PMD-sized and intermediate sizes for huge pages, limitations presented by the buddy allocator prevented their support for the time being.

2.5 State-of-the-art Linux Based Systems

Despite the promising performance benefits of huge pages, they seem to underperform in modern OSes [10] leading to many applications to advise disabling them [23]. This can be partially attributed to the cost-obliviousness of many MM huge-page policies. Linux Transparent Huge Pages (THP) [33, 24] greedy first-touch huge page allocation policies do not take into account if the huge page is utilized or if the expected benefits justify the setup costs. Over the past years, the following systems have been presented providing a more comprehensive approach to transparent huge page management.

2.5.1 Ingens

Ingens [18] identifies flaws in the huge page management strategies of popular Operating Systems and presents new policies to manage the related trade-offs. It addresses a number of huge page problems, with the most prominent being:

- Increased page fault latency cause by synchronous promotions
- Increased memory bloat and fragmentation
- Unfair provision of huge pages across processes

To tackle the above drawbacks it employs a number of strategies. First, it battles high page fault latency by disabling synchronous promotions. To implement huge page allocation, Ingens introduces a background kernel thread responsible of promoting base pages to huge pages.

Second, to alleviate memory bloat and fragmentation, Ingens implements dynamic, utilization driven huge page allocation policies. Specifically, it tracks through a bit-vector how many base pages of a huge page are in use. Only when this number surpasses an administrator-defined threshold will the memory region be backed with a huge page. Ingens also argues that this approach might be too conservative for a system with abundant physical memory contiguity. Thus it monitors the fragmentation of the system using the Free Memory Fragmentation Index (FMFI), quantifying whether the system is fragmented or not. When the system is un-fragmented Ingens copies Linux's aggressive huge page allocation policy. On the other hand, when the system is fragmented, it enables the utilization percentage threshold. This utilization based logic is also applied to huge page demotion decisions, where high-utilization huge pages are deferred from being demoted.

Finally, Ingens tries to achieve fairness in huge page promotions across processes. It describes memory contiguity as a valuable resource in a system, thus it distributes it equally among processes.

2.5.2 Hawkeye

Hawkey [25] aims to solve the same huge page management shortcomings presented by the Ingens paper, yet it follows a different approach. First, Hawkey acknowledges the high page fault latency introduced by synchronous huge page allocation, but it also argues that the synchronous approach avoids subsequent page faults in the memory region, increasing the response time of real-time applications. It recognizes that the majority of the page fault time is spent on zeroing out the allocated pages, thus, in order to minimize this latency and retain synchronous huge page support, it employs asynchronous page pre-zeroing through a background kernel thread. Subsequently, an already zeroed page may be used at fault time allocations.

Second, Hawkeye reasons that the utilization based approach is not sufficient to distinguish memory areas that would benefit the most from being backed by huge pages. Instead, it prioritizes huge page sized regions on metrics like recency, frequency and access-coverage (i.e., how many baseline pages are accessed inside a huge page), captured through finer-grained access tracking.

To provide fair provision of huge pages to processes, Hawkeye ranks applications based on their estimated MMU overheads, instead of their use of contiguous memory. It recognizes applications that exhibit high TLB pressure by measuring hardware performance counters (or by monitoring memory access patterns sacrificing accuracy for portability) and prefers them for huge page allocation.

Finally, Hawkeye tackles memory bloat under fragmentation, by scanning existing huge pages to identify zero-filled baseline pages within them. If the number of zero-filled baseline pages inside a huge page is beyond a threshold, the huge page is broken into its baseline pages and the zero-filled baseline pages are de-duplicated to a canonical zero-page through standard COW page management techniques. Contrarily to huge page allocation fairness, Hawkeye scans huge pages to demote starting from the applications with the lowest measured MMU overheads.

2.5.3 CBMM

CBMM (Cost-Benefit Memory Manager) [21] introduces a cost-benefit analysis model to kernel memory management (MM) policies in order to improve behavioral consistency in the memory subsystem.

It implements a new component in the Linux kernel, the estimator, along with cost-benefit models for three kernel MM policies: huge page management, asynchronous page pre-zeroing, and eager paging. Subsequently, the MM subsystem invokes the estimator at decisions points in the kernel code, places where decisions need to be made.

Specifically in huge page management, before promoting a series of base pages to a huge PMD-sized page, the estimator calculates the cost of the promotion, by monitoring the system's state and using empirically calculated fixed costs, and the benefit of it, leveraging user-provided offline-aggregated profiling data.

CBMM defines benefit of backing a memory region with huge pages as the cycles averted from TLB misses, and it quantifies it with the following algorithm: For each workload it:

1. Divide the address space into 100 different regions.
2. Choose a memory region.
3. Run the workload with only that region backed with huge pages.
4. Measure the performance speedup using as baseline an execution without huge pages.
5. Repeat steps 2 through 4 for every one of the 100 regions.

CBMM avoids using generalized heuristics that may not apply in different systems states and in various workload behaviours. Instead, its cost-benefit approach avoids costly operations that do not provide the required benefit, as long as the profiling information is not highly inaccurate.

Chapter 3

eBPF

The extended Berkeley Filter, or commonly known as eBPF [28, 8, 4, 12], is a revolutionary kernel technology that allows developers to load custom-written code into the kernel dynamically, changing the way it behaves. This way eBPF programs benefit from kernel's access to resources and system data, while the underlying infrastructure ensures security and efficiency. Thus, eBPF has found great success by allowing the user to enhance the kernel with new functionalities without the need to reboot the system, since no kernel modules are loaded and no change to the kernel source code is needed.

The kernel eBPF infrastructure ensures the safety of eBPF programs before being loaded to the kernel through an in-kernel verifier. The verifier ensures that an eBPF program will neither crash the machine nor lock it up in a hard loop, and it won't allow data to be compromised. If the program is not safe to run, the eBPF verifier will not allow it to be loaded into the kernel.

The execution of eBPF programs in the kernel follows an event-driven model. Upon loading an eBPF program into the kernel, the program is attached to an event. An example of such an event could be the entry to a system call like execve. From now on, whenever an execve system call is called, the attached eBPF program will be executed.

3.1 From BPF to eBPF

What is now known as "eBPF" originated from the BSD Packet Filter, which was originally described in a 1993 publication by Steven McCanne and Van Jacobson of Lawrence Berkeley National Laboratory [22]. The subject of this paper is a pseudomachine that is capable of executing programs called filters, which decide whether to accept or reject a packet on the network. The BPF instruction set, a general-purpose set of 32-bit instructions that closely resembles assembly language, was used to write these programs.

This early version, now referred to as "classic BPF" (cBPF), was integrated

into the Linux kernel by version 2.1.75 and quickly became a key tool in network packet capture utilities like tcpdump, offering an efficient way to capture packets to be traced out. Over time, the capabilities of BPF expanded far beyond packet filtering. The introduction of seccomp-bpf [9] in kernel version 3.5 marked a pivotal moment, enabling BPF programs to manage system call permissions and broadening its application to security contexts.

It was until kernel version 3.18, when "classic BPF" (cBPF) transitioned to its "extended BPF" (eBPF) state. This kernel version included a complete overhaul of the BPF instruction set to better suit 64-bit architectures, added new data structures called maps enabling efficient data sharing between kernel and user space, a new bpf() system call for interacting with eBPF programs from user space and an "eBPF verifier" kernel component that ensured safe execution of eBPF programs.

3.2 eBPF use cases

By allowing users to inject custom written programs in various points in the kernel, eBPF enabled the creation of advanced tools in performance monitoring, system observability, security and networking, while allowing the kernel to be fully programmable.

3.2.1 Tracing and Profiling

The eBPF technology revolutionized they way tracing was done in Linux systems. It leveraged a Linux feature named kprobes [16], that allows for traps to be set on almost any instruction in the kernel code, and similar tools like uprobes and tracepoints, in order to enable eBPF programs to attach virtually anywhere in the kernel and userspace application's code. With eBPF users can implement low overhead, fine grained profiling of the runtime behavior of both userspace and kernel space processes, giving unique and precious insights to troubleshoot system's performance issues.

3.2.2 Networking

In networking, developers can install load balancing policies, application profiling scripts, network monitoring procedures, and faster, more customized packet processing features using eBPF. eBPF is utilized by open-source solutions such as Cilium to offer safe and scalable networking for Kubernetes clusters, tasks, and additional containerized microservices. Furthermore, eBPF can expedite routing procedures and facilitate a faster overall network response by utilizing kernel-level package forwarding logic.

3.2.3 Security

Before eBPF, system security relied on specialized solutions focusing on various aspects of it. For instance, separate systems would be required for network-level filtering, process context tracing, and system call filtering. On the other hand, by expanding the fundamental skills of viewing and understanding all system calls and offering packet- and socket-level views of all networking processes, eBPF makes it easier to combine control and visibility over each component. This makes it possible to create security systems with enhanced control and greater awareness.

3.2.4 Scheduling policies

More recently eBPF made its way into the Linux scheduler [31]. A new scheduling class, named extensible scheduler class (or sched-ext), provides a framework that allows the implementation of a wide range of scheduling policies in BPF code. An application is then able to choose between default Linux scheduling policies and an eBPF scheduler that is loaded and running. This enables the development of tailor-made application specific schedulers, while also easing the experimentation of new scheduling policies with a plug-and-play model.

3.3 How does eBPF work?

3.3.1 The eBPF Virtual Machine

The Linux kernel supports the execution of eBPF programs by providing a sandboxed environment similar to a virtual machine. More specifically, eBPF programs are assembled from a set of BPF specific bytecode instructions acting on virtual, software implemented BPF registers. In order for these instructions to execute they need to be converted to the system's architecture specific instructions. Older systems achieved that using an in-kernel interpreter, mapping BPF instructions directly into machine code. The interpretation took place every time an eBPF program needed to run. The vast majority of modern systems have replaced this interpreter with a Just-In-Time compiler. This way, eBPF programs are compiled into native machine instructions just once, when the program is loaded into the kernel. This, along with compiler optimizations, provides a significant boost to eBPF programs' performance and also some hardening against Spectre related vulnerabilities within the interpreter.

3.3.2 The eBPF Instruction Set Architecture

The eBPF subsystem provides the developer a general purpose RISC-based instruction set. The eBPF instruction set has been originally designed in such

way that developers could write C-like programs that would compile into BPF bytecode instructions through a compiler back-end suite like LLVM.

Instruction Encoding

eBPF bytecode instructions are of a fixed size of 64 bits acting on virtual eBPF registers. They are compromised of an 8-bit opcode, two 4-bit fields representing the source and destination registers, a 16-bit offset for jump instructions and a 32-bit immediate value. Most instructions do not use all of the fields. The unused fields must be cleared to zero. Particular care has been given to the design of the eBPF instruction set to be easily mapped to common CPU architectures so that the step of interpreting or compiling from bytecode to machine code would be reasonably straightforward.

Instruction Classes

The eBPF instruction set architecture categorizes its instructions in 8 different classes based on the 3 least significant bits (LSBs) of the opcode field. These classes can be further grouped in three larger categories of Arithmetic and Logical Operations (supporting addition, subtraction, bit-wise operations, etc.), Load and Store instructions (moving data between registers and memory, supporting both direct and indirect addressing modes), Jump instructions (conditional and unconditional jumps for flow control) including Call instructions to support function calls, including helper function invocations provided by the kernel.

eBPF registers

The eBPF virtual machine uses 11 software implemented registers, numbered from 0 to 10. Registers 0 through 9 are used for general computation purposes, while register 10 is reserved as a stack frame pointer, which can only be read from and not written to. Register 0 usage is to store the return value from function calls and the exit value of eBPF programs. Registers 1 through 5 typically hold the arguments for function calls, while register 1, also conventionally holds the context of the eBPF program. Subsequently, these registers, characterized as "scratch", do not guarantee that their value will remain untouched between function calls. If the callee would like to save their value they should either push them to the eBPF stack or save them to callee saved registers. These conventions imply an upper bound of 5 input arguments per function call. Registers 6 to 9 are callee saved registers that will be preserved across function calls.

eBPF context

What we referred to as "context" acts as the input argument for an eBPF program. Depending on program type the context is different, for example, in

a type XDP networking eBPF program register 1 might point to a kernel data structure specific for representing the received network packet. It usually provides the eBPF program a set of useful information regarding the event it is attached to.

3.3.3 Attaching to an Event

As already mentioned eBPF programs follow an event-driven execution model, so in order to execute they must be attached to some kind of event. Programs can have lots of different purposes, they can for example record information, modify information, make decisions and cause side effects. Where a program is allowed to attach and what is allowed to do depends on its program type.

A feature called kernel probes [16] (kprobes) had existed in the Linux kernel since 2005, enabling developers to trap almost any kernel code address, except some parts of the kernel code that are not allowed to be trapped. That means that a kprobe can be inserted on virtually any instruction in the kernel. (kernel docs for kprobes)

BPF_PROG_TYPE_KPROBE is a type of eBPF program that can be attached to kprobes. It is evident that kprobes and eBPF programs work very well together, since they allow a developer to attach an eBPF program virtually anywhere in the kernel.

3.3.4 Verifier

The importance of the eBPF technology has been established, allowing developers to inject custom written programs into the kernel, enabling the kernel to be extended at runtime. The verifier [5] component of the eBPF subsystem's stack is responsible for ensuring the safety of eBPF programs, by enforcing restrictions and security checks to their execution. Execution of unverified code running in kernel mode imposes serious security threats like corrupting memory and leaking sensitive information to malicious third parties, as well as causing the kernel to crash or to be trapped in deadlock. Inline security checks at runtime introduce non acceptable overheads, thus verification of eBPF programs through static analysis at load time is crucial in order to meet performance quota. This is a trade-off between ease of use and performance, since developers are called to tailor their eBPF programs to pass the strict verifier checks so that they can run at native speed.

Important Verification Checks

Let's take an in depth look at what criteria must be met for a program to be considered safe from the verifier's perspective.

Program Termination First of all, eBPF programs must always terminate, so the verifier rejects any program that could contain infinite loops or recursions. Traditionally this was achieved by rejecting any loop instruction all together, but in newer kernel versions bounded loops have been introduced for developer's comfort. The verifier, at least at this time of writing, is able to process up to 1 million instructions of eBPF programs, thus hard coding that all programs must eventually terminate. Any program with more instructions than the upper limit is going to be rejected.

Memory Access Furthermore, in order to battle the leak of sensitive information, the verifier prohibits programs to read arbitrary memory. The verifier provides programs a controlled way of accessing memory through a collection of verified functions called helper functions. Different program types have access to a different set of helper functions, for example, tracing programs are allowed to use functions that can read memory areas. The verifier is responsible for ensuring the correct usage of these helper functions, and that programs have the appropriate license to use them (only GPL compatible programs can use eBPF helper functions licensed under GPL). Tracing programs require root privileges in order to be loaded in the kernel and thus are not a security risk. Another restriction on the memory side is that programs are not permitted to read uninitialized memory.

Pointer de-referencing A big part of the verifier is dedicated to ensure safe pointer dereferencing. De-referencing a pointer that points to an invalid location could either leak sensitive information, or crash the program entirely, in the case of a null pointer. To avoid that, the verifier employs a technique called range analysis verification, where it tracks the bounds of the values of each register, in order to reject any out of bound access to memory through pointer arithmetic. To pass this verification step, programmers ought to check whether a pointer is NULL or out-of-bounds before de-referencing it.

Context Information As already mentioned, context information is passed in every eBPF program as an input argument. Context information is usually a pointer to some data structure relevant to the program and the attachment type. For example, in BPF_PROG_TYPE_KPROBE type programs, the context is a pointer to a data structure resembling a state copy of the CPU registers at the time of the probed instruction. Not all eBPF programs are allowed to access every field of this data structure, therefore the verifier ensures that each eBPF program accesses only context information that is allowed to.

Concurrency Also, to avoid system deadlocks, programs require the systematic approach to locking to ensure correctness under concurrency. Each lock held by a program must be released, and a program is not allowed to hold more than one lock at a time.

This concludes some of the most important conditions, among others, that must be met in order for an eBPF program to pass the verifier. Now we are going to focus on how the verifier guarantees that none of these constraints are breached in the entirety of a program.

Verification Process

The basic premise is that the verifier checks every possible permutation of a program mathematically. This is achieved by walking the code from the beginning and constructing step by step a graph based on branch instructions. Starting from the first instruction and descending all possible paths, the verifier simulates the execution of each instruction and keeps track of the state changes of every register. The state of each register is represented by a structure named bpf_reg_state, which includes fields describing what type of value is held in that register along with its minimum and maximum value. Each time a branch instruction is encountered the verifier will push a copy of the current state of the registers onto a stack and will continue exploring one of the possible execution paths. When the end of the program is reached, another execution path is popped from the stack to be evaluated. This is done until the end of the program is reached and the stack is empty (the program passes the verifier), or an invalid operation is performed (the program does not pass the verifier).

Checking each possible execution path of a program can get computationally expensive, subsequently the verifier is optimized with state pruning, a technique aiming to avoid reevaluating paths that are essentially equivalent. This is implemented by storing the state of specific registers every fixed number n of instructions. The state of which registers to store is determined through an algorithm called “Register Liveness Tracking”. This algorithm detects which registers are necessary for the verification of subsequent executions of the program. If the verifier later arrives at the same instruction with a state that matches the previously cached state, the verification process stops for the current path as it is known to be valid.

3.3.5 eBPF Helper Functions

Taking into account the restrictions imposed by the verifier, an eBPF program on its own is quite limited. By default, eBPF programs can perform basic operations like reading from and writing to a local stack, performing mathematical operations on registers, making conditional jumps, and calling internal functions. On the other hand, they are neither allowed to directly call to any kernel function (unless it has been registered as a kfunc), nor access kernel information. These capabilities are crucial for most practical applications of eBPF programs.

In order to enable eBPF programs to interact with system resources in a safe manner, the Linux eBPF subsystem provides a list of predefined ”helper functions”. These helper functions act as an interface between the eBPF program and

Table 3.1: eBPF Register Value Types

NOT_INIT	Register value is not initialized
SCALAR_VALUE	Register is not a valid pointer
PTR_TO_CTX	Register points to input context
CONST_PTR_TO_MAP	Register points to a program defined map
PTR_TO_STACK	Register is the frame pointer
PTR_TO_MAP_VALUE	Registers points to a map element value
PTR_TO_MAP_KEY	Registers points to a map element key
PTR_TO_MEM	Registers points to a valid memory region
PTR_TO_FUNC	Registers points to a bpf program function

the kernel. They serve a wide range of purposes, from printing debugging messages to accessing kernel memory -e.g. interacting with eBPF maps, manipulating network packets.

Each type of eBPF program operates in a different context, which means that not all helper functions are available to every eBPF program. Instead, each program type has access to a specific subset of helpers that makes sense to it. For example, network-related eBPF programs might have access to functions that manipulate packet data, but a helper retrieving the process's ID is not available, since no process is related to a packet capture.

Helper functions act as an interface between the eBPF program and the kernel. This design ensures that eBPF programs remain portable and do not break across kernel versions.

Internally, eBPF programs call these helper functions directly, without requiring any foreign-function interface or intermediary, which means that invoking a helper introduces no additional performance overhead. This direct calling mechanism ensures that eBPF programs remain highly efficient.

3.3.6 eBPF Maps

A map [12] is a data structure that is accessible from both userspace and from an eBPF program. One of the most substantial things that sets extended BPF apart from its traditional predecessor are maps. Maps can be used to exchange data between different eBPF programs or to facilitate communication between eBPF code operating in the kernel and a user space application. Interaction with eBPF maps from user-space is facilitated through a dedicated BPF system call interface. Maps can be accessed with eBPF programs using provided helper functions. They provide a persistent storage layer that is commonly used for:

- User space writing configuration information to be retrieved by an eBPF program.

- Storing state across execution of different (or the same) eBPF programs.
- Writing results or metrics of eBPF programs, in order to expose them to userspace applications.

The eBPF subsystem defines various types of maps, each one suited for different purposes, but in general they are all key-value stores. Some map types are defined as arrays, which always have a 4-byte index as the key type; other maps are hash tables that can use some arbitrary data type as the key. Other map types are optimized for particular types of operations, such as first-in-first-out queues, first-in-last-out stacks, least-recently-used data storage, longest-prefix matching, and Bloom filters (a probabilistic data structure designed to provide very fast results on whether an element exists).

Some eBPF map types hold information about specific types of objects, like sockmaps and devmaps that hold information about sockets and network devices and are used by network-related eBPF programs to redirect traffic. A program array map stores a set of indexed eBPF programs, and is used to implement tail calls, where one program can call another. There's even a map-of-maps type to support storing information about maps.

In order to address concurrency issues, where a map is simultaneously accessed by different CPU cores, eBPF supports spinlocks for some types of maps.

Chapter 4

Implementation

In the previous chapter, we presented the eBPF technology. We described how it works internally and how modern systems take advantage of it in a plethora of use cases.

In this chapter, we present how we leveraged this new technology to achieve our main objective; enhance the decision-making process of the Linux memory management subsystem regarding huge page allocation. Our system enables users to provide profiling information to an eBPF program, able to guide the kernel's selection of the most suitable huge page size for a memory region at fault time.

While the eBPF program we developed can be configured to choose between an arbitrary selection of huge page sizes, we only target 64KiB and 2MiB huge pages. This decision was made because Linux mTHP does not support 32MiB huge pages, while the underlying hardware used for the evaluation benefits from 64KiB huge pages.

In the next sections we are going to provide a high-level overview of our implementation, and then delve deeper into more technical details.

4.1 High-Level Overview

This section describes how our eBPF system makes the memory management subsystem modular. Specifically, we explain the code path of our system at page fault time and how an attached eBPF program can influence it with hints from user space. We also briefly explain the core components that comprise our system.

4.1.1 Hooks on Page Fault Path

In order to execute, every eBPF program needs to be attached to a specified attachment point. The capabilities of eBPF programs are greatly influenced from the place of the attachment point and the context it provides to eBPF programs. Subsequently, at first, we implement a hook point for our eBPF program in the Linux page fault path using the kprobes technology. We place our hook point right

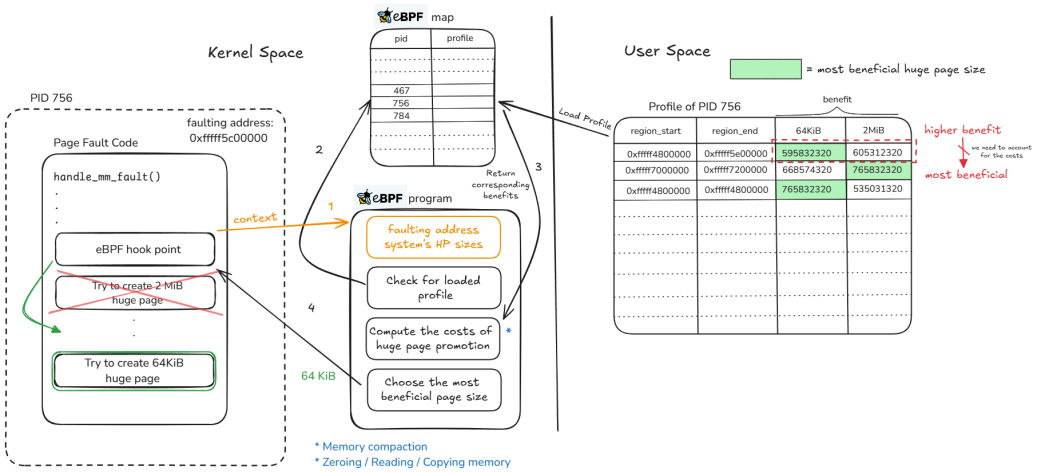


Figure 4.1: ebpf-mm: High-Level overview of our eBPF-enhanced huge page management

before the decision of the Linux kernel to promote a faulting memory region to a huge page or not. When the kernel code path reaches the hook point the eBPF program will get triggered and start executing. Its actions will determine whether or not this decision will be made.

4.1.2 Context

As already mentioned the context information passed to an eBPF program acts like an input for it. It is crucial to pass important information to the eBPF program if we expect to make any informed decision with it. In our case, we need to distinguish a page fault from others, in order to provide tailored to this fault profiling information. To achieve that, we provide to the eBPF program the address of the memory operation that caused the page fault handler and the available huge page sizes of the faulting memory region to promote to.

4.1.3 Search for a profile

Our system stores per-process profiling information associated with the benefit of different huge page sizes inside an eBPF map. The eBPF map uses as key a process's ID and as value a data structure containing profiling information for memory regions of the corresponding process. The eBPF program proceeds to retrieve the profile associated with the faulting process. In the case it does not find an entry with the specific process ID, or the entry contains a NULL profile, then normal Linux behaviour resumes. After finding a profile for the faulting process the eBPF program continues by searching the profile for the first memory range that the faulting address belongs to. If it finds a profiled memory range, it

retrieves the benefit for the different huge page sizes, if not it falls back to normal Linux behaviour.

User-space program We developed a complimentary user-space program that is responsible for loading and unloading profiles from the eBPF map. It takes two arguments as input parameters. A file with a specific structure, containing profile information for a range of memory regions. A process ID that indicates to which process the profile corresponds to.

Profile Structure The profiles are structured in a specific way. Each line of the profile describes the benefit of all huge page sizes about a specific memory region defined with a start and end address. Specifically, it contains a number of comma separated values. The first two values signify the start and end address of the region accordingly. The remaining values represent the benefit of using each huge page order in ascending order (e.g. the first value is the benefit for order 2 huge page).

4.1.4 Compute the costs of huge page promotion

As previously mentioned the promotion of memory regions to huge pages has both performance benefits and associated costs. The cost of promoting a memory region to a huge page is non-trivial. Our eBPF program considers two main contributing factors to the overall cost; the time needed to find a large enough contiguous block of physical memory and the time needed to prepare it.

Physical Memory Contiguity and Compaction In systems with ample free memory, finding a contiguous block of physical memory is a relatively inexpensive operation, hence in this case we consider this cost negligible. However, in systems where memory is fragmented, the kernel may need to perform compaction to free up enough contiguous memory. Memory compaction involves relocating active pages to create contiguous space, which can be time-consuming and introduces additional system overhead. In our model, we enable eBPF programs to monitor the current fragmentation state of the system in real-time, through a newly introduced helper function.

Page Preparation Before allowing users to perform operations on allocated memory the operating system needs to make sure to prepare the underlying physical pages, introducing additional costs. For anonymous memory regions, this preparation includes zeroing the memory before use. For file-backed memory, the actual data may need to be loaded from disk, introducing additional I/O latency. Our eBPF implementation targets only anonymous memory allocations, thus we only take into account the zeroing cost.

Fixed Cost Estimation In our system, the costs associated with memory compaction and page preparation are estimated empirically as previous works [21]. We observe the system during different states of fragmentation and use this data to calculate a fixed cost for compaction and page preparation. These costs are then used in real-time by the eBPF program to determine whether huge page promotion is worthwhile.

4.1.5 Hint the kernel on beneficial page sizes

The next step of our eBPF program is to compare the preparation costs with the expected profiled benefit for different huge page sizes. We use the available page orders provided as context to our eBPF program to determine two things; which of these are beneficial and which one is the most beneficial. Then we hint the kernel to try to back the faulting memory region with the most beneficial huge page order. If this fails, the kernel is able to promote to a different huge page order that is smaller than the one tried. Thus, we also hint the kernel which orders smaller than the optimal choice are still beneficial to promote to.

4.2 Technical Implementation Details

4.2.1 Hooks & Context

In order to implement the hooks on the page fault path we leveraged the kprobes technology.

kprobes Kprobes are an instrumentation tool, allowing a programmer to establish a breakpoint on virtually any instruction in the kernel. When a CPU hits a breakpoint instruction (commonly optimized as a jump instruction) the registers are saved and the codepath deviates executing a specified pre-handler (before the probed instruction) and post-handler (after the probed instruction) function. Kprobes also allow the handler functions to access the CPU registers before the call instruction, through a data structure named pt_regs. The bpftool tool automates the process of registering a kprobe and attaching the eBPF program as a pre-handler. The libbpf library automates the process of getting the input arguments of a probed kernel function through the pt_regs data structure with the macro BPF_KPROBE.

To implement the hook in the page fault path we define a new function in the kernel code. The input arguments of this kernel function will act as context information for the attached eBPF programs. Thus, we provide this function a pointer to a data structure as the first argument, which contains the faulting address of the page fault along with the available huge page orders available for the memory region it belongs to. The body of this function prints some debugging

information to the kernel's debug log, which can be omitted, and then simply returns. We then strategically call this function in two points in the page fault path, which is where our eBPF program will be attached through a kprobe. We need to make sure that the compiler does not inline this function, because kprobes do not work with in-lined functions.

There are two points from where our eBPF program can be triggered. We make this decision in order to interfere as little as possible with the default Linux behaviour. The first point of triggering the eBPF program is right before the Linux kernel tries to back the memory region with 2 MiB, PMD-based huge pages. The second point is right on the mTHP decision point, where the Linux kernel iterates all the available mTHP page sizes from bigger to smaller trying to back the region with said size each time.

4.2.2 Interacting with the kernel

In the previous section we frequently described utilities our eBPF program is implementing, which in order to complete them it would need to interact with the kernel.

We can distinguish these interactions in two categories. The first category explains how both our eBPF program and the user-space program interact with the defined eBPF map, in order to retrieve and store profile data. The second category has to do with how the eBPF program interacts with kernel memory in order to access its context, get information about the state of the system and alter the available huge page orders for a faulting memory region.

In the previous chapter we stated that, for security reasons, the eBPF verifier severely limits such capabilities from eBPF programs. In order to get around these limitations, while ensuring eBPF's security we used both existing eBPF helper functions and developed new ones.

eBPF map

The eBPF map holding the per-process profiling information is undoubtedly a core component of our system. Let's take a closer on how we define this map and how eBPF implements it.

```
1 struct {
2         __uint(type, BPF_MAP_TYPE_ARRAY);
3         __type(key, u32);
4         __type(value, struct profile_t);
5         __uint(max_entries, 10240);
6         __uint(pinning, LIBBPF_PIN_BY_NAME);
7 } my_map SEC(".maps");
```

The above code defines the attributes of the eBPF map we are using. We observe in the body of the defined structure several key things:

- Our map is of type BPF_MAP_TYPE_ARRAY. This type of map is a generic map type similar to simple arrays in C. It is indexed with a numeric key starting at 0 and incrementing, while eBPF imposes no restrictions on the structure of the value. A possible optimization of this would be to use a map type of BPF_MAP_TYPE_HASH, ensuring faster accesses to the profile data.

- We are forced to use a key size of 4 bytes, essentially a 32-bit unsigned integer. Since process IDs are also 32-bit unsigned integers, this doesn't add any complexity.

- We define the type of the value as a profile_t structure. This is a structure we defined in order to represent profiling data. This approach imposes a restriction that the profile structure is of fixed size. This means that every process is only able to profile a predefined number of ranges. We can overcome this limitation by implementing a different type of eBPF map of type BPF_MAP_TYPE_ARRAY_OF_MAPS, where each value can be a different map of varying size. For sake of simplicity we did not choose that approach.

- We need to define the maximum number of entries our eBPF map will use. We set this number at 10240, considering that there is no point of profiling over that number of processes at the same time.

- The last configuration option is to pin our eBPF map by name. This option creates a file in a bpf pseudo-filesystem with the file descriptor representing our eBPF map structure. This serves two purposes. First, the user space program responsible for loading the profiles is able to easily find the eBPF map. Second, the eBPF map is not going to be de-allocated after the eBPF program is terminated for some reason, since a reference to it will still exist.

- The use of the SEC() macro in last line after the definition of the map structure declares an ELF section in the compiled ELF object. Specifically, it instructs the compiler to put additional metadata for our defined map in the .maps ELF section. This is later used by the libbpf library to generate the BTF information related to the structure of the map, which will be used in the deployment of the eBPF program, described in a later section.

The profile_t structure As already described the profile_t structure represents the profile data for a process.

```
struct cb_struct {
        unsigned long long benefit;
        unsigned long long cost;
};

struct prof_range {
        unsigned long long start;
        unsigned long long end;
        struct cb_struct cb[MAX_HP_ORDERS];
};

struct profile_t {
        int enabled;
        struct prof_range ranges[NUM_RANGES];
};
```

We observe that the profile structure contains two fields. The first one is an integer named enabled used to unload a profile from the kernel easily and efficiently. Our eBPF program will ignore any profile with the enabled field set to 0. The next field is a fixed size array containing a structure representing a profiled memory range. The structure for the profiled memory range contains the start and end address of it, as well as an array containing the expected benefit for the usage of each huge page order.

Access the map Since, an eBPF map is stored in kernel memory, the eBPF verifier does not allow eBPF programs to access it with arbitrary pointers. The kernel provides access and manipulation of eBPF maps through specific commands of the bpf() system call. Here, we present the already libbpf helper functions our eBPF and user-space programs used in order to access the eBPF map that act as a wrapper to the underlying system call commands.

- `bpf_map_lookup_elem()`

  This helper function performs a lookup in a map for an entry associated to key. The map argument must be a pointer to a map definition and key must be a pointer to the key you wish to lookup. The return value will be a pointer to the map value or NULL. The value is a direct reference to the kernel memory where this map value is stored, not a copy. Therefor any modifications made to the value are automatically persisted without the need to call any additional helpers.

- `bpf_map_update_elem()`

  This helper function is used to write values to maps. Arguments of this helper are map which is a pointer to a map definition, key which is a pointer to the key you wish to write to, value which is a pointer to the value you wish to write to the map, and flags which are described below. The flags argument can be one of the following values:

  - BPF_NOEXIST - If set the update will only happen if the key doesn't exist yet, to prevent overwriting existing data.
  - BPF_EXIST - If set the update will only happen if the key exists, to ensure an update and no new key creation.
  - BPF_ANY - It doesn't matter, an update will be attempted in both cases.

  The libbpf library provides an equivalent function for user-space programs, but instead of providing a reference to a map definition, a user needs only to provide a file descriptor to it. Our user-space programs leverages this libbpf API function to load the profile to the eBPF map.

- `LIBBPF_API bpf_obj_get()`

  This is a wrapper for the bpf() system call called with the BPF_OBJ_GET command, provided by the libbpf library. Specifically it takes as an argument the path to a pinned bpf object and return a file descriptor to it. Our user-space program uses this helper functoin to get a file descriptor for our eBPF map.

Hint the Kernel about beneficial Huge Page Orders

As for now we have explained in technical detail how our eBPF program accesses profile data, provided by user-space, containing the expected benefit of a huge page order. This benefit is then compared to a system-specific cost in order to decide which of the available huge page orders for the faulting memory region are beneficial. The final step of our eBPF program is to return to the kernel the beneficial huge page orders. Since we need to interact with the kernel in a specific way we developed two new helper functions:

- A helper function that informs the eBPF program about the system's state regarding free memory pages. This helper function enables the eBPF program to determine the cost for setting up a huge page for a memory region.
- A helper function designed to pass the beneficial huge page orders to the kernel.

bpf_free_huge_page_status(int order) This helper function is used to determine if the system has any available physical page of order equal or greater than the one provided as an argument. If there is a free page of the requested order we assume that the huge page allocation of the corresponding size will not induce any additional costs for finding a large enough contiguous block of memory (compaction or reclamation). In that case we consider that the sole cost of setting up the huge page is the time it takes to zero its memory contents.

```
BPF_CALL_1(bpf_free_huge_page_status, int test_order) {
        int zone_idx;
        struct zone *zone;
        struct page *page;
        struct free_area *area;
        bool is_free = false;
        int order;
        unsigned long flags;

        pg_data_t *pgdat = NODE_DATA(numa_node_id());
        for (zone_idx = ZONE_NORMAL; zone_idx <
           MAX_NR_ZONES; zone_idx++) {
                zone = &pgdat->node_zones[zone_idx];

                for (order = test_order; order <
                   MAX_ORDER; ++order) {
                        area = &(zone->free_area[order])
                        is_free = area->nr_free > 0;

                        if (is_free) {
                                return 1;
                        }
                }
        }

        return 0;
}
```

bpf_update_orders(struct mm_action *action, unsigned long orders) This helper function takes two arguments as input. The first one is a pointer to the data structure that represents the context information we provide the eBPF program. The second one is an unsigned long integer representing the huge page orders we want to store to the context data structure. Before the kernel invokes our function used as a hook point, it stores inside this data structure the available huge page orders. After the eBPF program is done executing and the execution

flow returns back to the kernel code, the kernel updates the available huge page orders for this fault with the ones stored in our data structure. If our eBPF program provided different huge page orders, the kernel will be informed. In any other case the kernel will preserve its default available huge page orders.

```
BPF_CALL_2(bpf_update_orders, struct mm_action *, action
    , unsigned long, bpf_orders) {
        action->hugepage_orders = bpf_orders;
        return 0;
}
```

4.2.3 Calculating Fixed Costs & Benefit

Costs

In order to calculate the costs for a specific kernel operation, we leverage the interface of Linux Kernel Tracepoints. In the following paragraphs we present the results from observing our system during a preliminary execution of our workloads used for evaluating it. Specifically, we placed tracepoints in crucial points in the kernel code, measuring the number of cycles it took for the following operations.

Zeroing a 2MiB huge page Here we calculate the cost of zeroing a 2MiB huge page. We added the following tracepoint:

```
#ifdef CONFIG_PFTRACE
    cycles = get_cycles();
#endif /* CONFIG_PFTRACE */

    clear_huge_page(page, vmf->address, HPAGE_PMD_NR);

    __folio_mark_uptodate(folio);
#ifdef CONFIG_PFTRACE
        if (tracepoint_enabled(zerohugepmdpage)) {
                do_trace_zerohugepmdpage(get_cycles() -
                   cycles);
        }
#endif /* CONFIG_PFTRACE */
```

We observe that the time it takes for the Linux kernel to zero a 2MiB huge page in our experimental machine is approximately 1000 cycles.

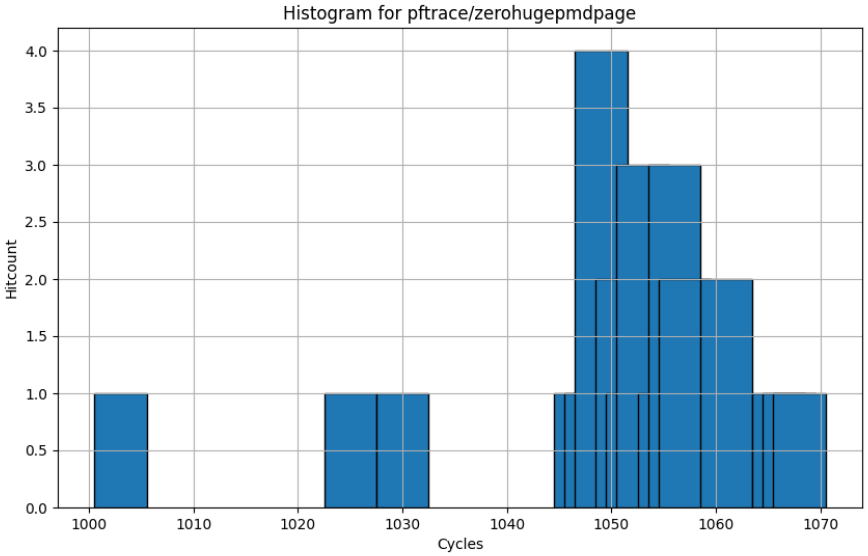


Figure 4.2: 2MiB page zeroing cost

Zeroing a 64KiB huge page Here we calculate the cost of zeroing a 64KiB huge page. We added the following tracepoint:

```
#ifdef CONFIG_PFTRACE
    cycles = get_cycles();
#endif /* CONFIG_PFTRACE */
    clear_huge_page(&folio->page, vmf->address, 1 <<
       order);

#ifdef CONFIG_PFTRACE
    if (tracepoint_enabled(zeromthppage)) {
        do_trace_zeromthppage(get_cycles() - cycles);
    }
#endif /* CONFIG_PFTRACE */
```

We observe that the time it takes for the Linux kernel to zero a 64KiB huge page in our experimental machine is approximately 50 cycles.

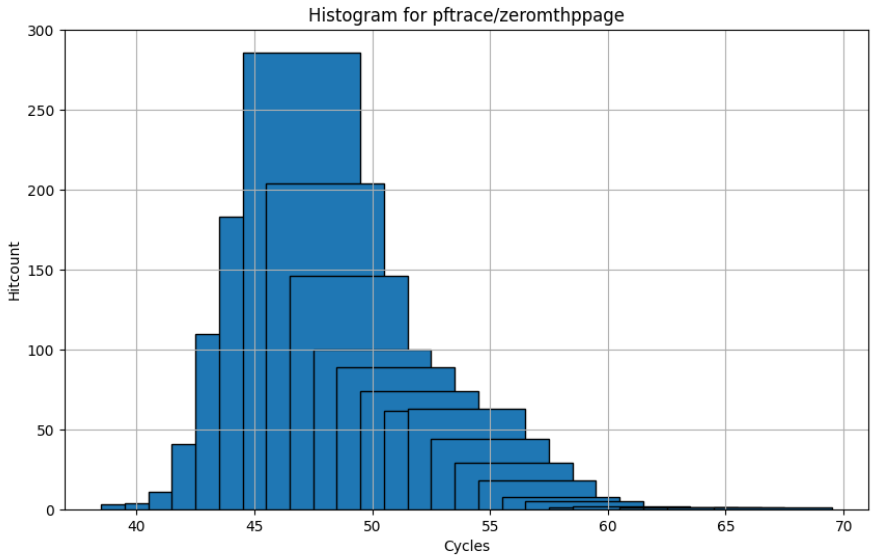


Figure 4.3: 64KiB page zeroing cost

Compaction & Page Reclamation To measure the cycles our system needs to perform the memory compaction and reclamation operations we installed the following tracepoints. Due to the rarity of these operations in our system, we were unable to gather sufficient data for a robust cost estimation. However, as previous works that have measured some parts of the costly page walk involving compaction have stated [21], we deem them incredibly expensive. Thus we assign them a very large fixed cost, that would outweigh the expected benefit almost every time.

```
#ifdef CONFIG_PFTRACE
    cycles = get_cycles();
#endif /* CONFIG_PFTRACE */

        psi_memstall_enter(&pflags);
        delayacct_compact_start();
        noreclaim_flag = memalloc_noreclaim_save();

        *compact_result = try_to_compact_pages(gfp_mask,
            order, alloc_flags, ac, prio, &page);

        memalloc_noreclaim_restore(noreclaim_flag);
        psi_memstall_leave(&pflags);
        delayacct_compact_end();

```

```
#ifdef CONFIG_PFTRACE
    if (tracepoint_enabled(runcompaction)) {
        do_trace_runcompaction(get_cycles() - cycles);
    }
#endif /* CONFIG_PFTRACE */
```

```
#ifdef CONFIG_PFTRACE
    cycles = get_cycles();
#endif /* CONFIG_PFTRACE */
        psi_memstall_enter(&pflags);
        *did_some_progress = __perform_reclaim(gfp_mask,
            order, ac);
        if (unlikely(!(*did_some_progress)))
                goto out;

retry:
        page = get_page_from_freelist(gfp_mask, order,
           alloc_flags, ac);

        /*
         * If an allocation failed after direct reclaim,
             it could be because
         * pages are pinned on the per-cpu lists or in
            high alloc reserves.
         * Shrink them and try again
         */
        if (!page && !drained) {
                unreserve_highatomic_pageblock(ac, false
                   );
                drain_all_pages(NULL);
                drained = true;
                goto retry;
        }
out:
        psi_memstall_leave(&pflags);
 #ifdef CONFIG_PFTRACE
    if (tracepoint_enabled(runreclaim)) {
        do_trace_runreclaim(get_cycles() - cycles);
    }
#endif /* CONFIG_PFTRACE */
```

4.2.4 eBPF deployment

In the previous chapter we described the execution model of eBPF programs. Specifically, we mentioned that the kernel accepts eBPF programs written in eBPF's bytecode instructions, runs the verifier to ensure safety, JIT compiles the program into machine code that runs natively on the target CPU, attaches them to an event that when triggered, will trigger execution of the eBPF program.

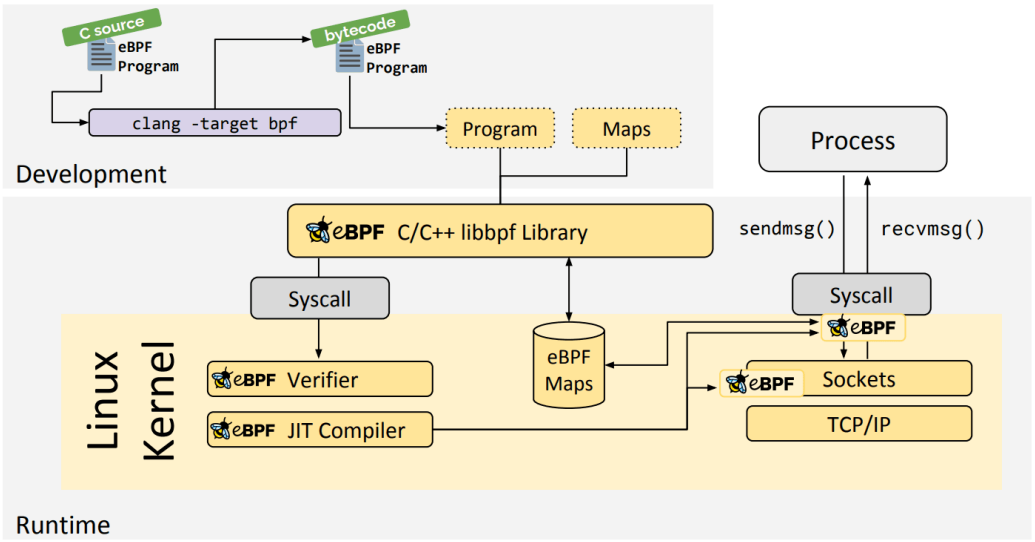


Figure 4.4: eBPF program's life cycle

Here we describe the process and tools we used for setting up our eBPF system.

The first step is using the bpftool utility in order to generate a header file named vmlinux.h for our eBPF program, that contains all the data structure information about the running kernel it might need.

Next, we need to compile our eBPF program, written in C, into eBPF bytecode. This is done using the clang compiler from the LLVM project target the eBPF instruction set with the -target bpf flag, which produces a compiled eBPF object file.

We leverage the bpftool utility again, in order to generate a BPF skeleton header file from the previously compiled eBPF object. The skeleton contains higher level abstractions of the libbpf library that the user space code can use to manage the life-cycle of the eBPF program.

Finally, we develop the user space program, built with libbpf, responsible for loading the eBPF program into the kernel, setting up and attaching it to the specified event using the following functions:

- `check_estimator_bpf__open_and_load()`

  This function uses libbpf to perform tasks such as setting up eBPF maps, loading the bytecode into the kernel, and verifying the program with the kernel's BPF verifier.

- `check_estimator_bpf__attach()`

  This function sets up a kprobe (in this case in the call instruction of our mm_estimate_changes function in the kernel) and registers it as a perf event. It continues by attaching the eBPF program in said event, allowing it to execute whenever it is triggered. The output of the eBPF program can be observed through tools like trace_pipe.

Makefile:

```
TARGET = check-estimator
ARCH = $(shell uname -m | sed 's/x86_64/x86/' | sed 's/
   aarch64/arm64/')

BPF_OBJ = ${TARGET:=.bpf.o}
USER_C = ${TARGET:=.c}
USER_SKEL = ${TARGET:=.skel.h}

all: $(TARGET) $(BPF_OBJ)
.PHONY: all

# Build user-space code

$(TARGET): $(USER_C) $(USER_SKEL)
        gcc -Wall -o $(TARGET) $(USER_C) -L/root/libbpf/
           src -l:libbpf.a -lelf -lz

# Build BPF code

%.bpf.o: %.bpf.c vmlinux.h
        clang \
            -target bpf \
        -D __TARGET_ARCH_$(ARCH) \
            -Wall \
            -O2 -g -o $@ -c $<
        llvm-strip -g $@

# Build skeleton

```

```
28 $(USER_SKEL): $(BPF_OBJ)
29         bpftool gen skeleton $< > $@
30
31 vmlinux.h:
32         bpftool btf dump file /sys/kernel/btf/vmlinux
              format c > vmlinux.h
33
34 clean:
35         - rm $(BPF_OBJ)
36         - rm $(TARGET)
```

Chapter 5

Experimental Evaluation

5.1 Experimental Setup

We implement our system for Linux v6.9 and evaluate it on Ubuntu 22.04. We run our experiments in a virtualized environment using KVM [17] hypervisor and QEMU [6] emulator. This environment executes on an Ampere Altra [1] server host, with 2 nodes of 80 ARMv8-2A+ Neoverse N1 [3] cores, each with 256GiB of memory. The MMU includes separate data and instruction fully-associative L1 TLBs of 48 entries each, and a unified 5-way set-associative L2 TLB with of 1280 entries of any size. L1 misses cost on average 3 cycles and L2 misses over 15 cycles. We use a single NUMA node and pin each thread on a single core. For the omnetpp workload we replace GNU libc's malloc with gperftools tcmalloc [11].

5.2 Benefit Profiling Methodology

The benefits of huge page promotion primarily stem from the reduced overhead of address translation caused by improved TLB performance. Huge page increase the TLB reach by reducing the number of TLB entries required to map large regions of memory, which in turn reduces the frequency of costly page table walks. However, the benefit of huge pages depends heavily on the access patterns of the application and the size of its working set.

We define the benefit of using a huge page order for backing up a memory region as the number of cycles averted due to TLB misses in case the region was backed by base pages. The calculation of the exact number of averted TLB cycles seem to be cumbersome, since ARMv8 doesn't support page table walk cycles PMUs. Further, the cost of a single TLB miss relies on the latency to perform a page table walk, which can fluctuate due to the state of internal MMU caches, or where in the memory hierarchy the page tables reside.

Previous work [20] characterizes memory regions of applications as beneficial or not for huge pages, characterizing their memory access pattern through a metric

called page reuse distance -i.e. the number of memory accesses to different pages that occur between consecutive accesses to the same page (whether it's 4KiB, 64KiB, or 2MiB).

Specifically three categories are recognized:

- **TLB-friendly**

  The memory accesses in this memory region are characterized by high spatial and/or temporal locality (e.g. sequential accesses). Even though the page reuse distance may be average, the high frequency of memory accesses among a page keeps the ratio of TLB misses per memory accesses low. Base pages of 4KiB prove to be sufficient for such regions with huge pages providing little additional benefit.

- **High-Reuse TLB-Sensitive Accesses**

  The memory accesses in this memory region exhibit generally low spatial and temporal locality. In the scenario where 4KiB base pages are used memory accesses trigger a high amount of TLB misses, since the page reuse distance is too high to keep the entries in the TLB hierarchy. On the other hand, the pattern of these memory accesses exhibit good locality and low page reuse distance on the huge page granularity. These are the memory regions we are trying to identify, since they prove to be the most beneficial for promoting to huge pages.

  Memory access patterns that fall in the edge of this and the previous categories could benefit most from an intermediate huge page size like 64KiB, supported by the arm processor through the contig-bit.

- **Low-Reuse TLB-Sensitive Accesses**

  The memory accesses in this memory region exhibit such low locality that even promoting to a huge page wouldn't provide any significant benefit. This could indicate sparse and low in frequency memory accesses. If the memory accesses in this memory region are frequent a bigger than 2MiB huge page could be more sufficient.

Instead of calculating the specific true cycles averted, we use technologies like DAMON and arm hardware performance monitors to measure the access frequency of specific memory regions, and how many TLB misses they induce. We use the above information as proxy to determine in which category each memory region belongs to.

DAMON First, we monitor the memory access patterns of the application using the DAMON [27] (Data Access Monitor) framework. We configure DAMON to provide feedback on the frequency of memory accesses at a granularity that matches the huge page size. By observing the access frequency of memory regions,

we can estimate the benefit of backing those regions with huge pages. Regions of 2MiB with high access frequency prove potential candidates for huge pages, as they reduce the number of TLB misses and page table walks.

Hardware Performance Monitors As previously stated, we also take into account the TLB misses memory accesses to a 2MiB region induce, other than their frequency. We estimate the TLB misses through hardware assisted TLB miss sampling with ARMv8-A's Statistical Profiling Extension (SPE) [2].

5.3 Experiments

We evaluate our system with two SPEC CPU 2006 benchmarks [13] (astar and omnetpp) and a micro-benchmark specifically developed characterizing the needs of a workload that benefits the most from our eBPF system implementation.

We run these benchmarks in three different configurations:

- Linux using only 4KiB base pages (THP turned off)
- Linux with 2MiB huge pages (THP turned on)
- eBPF-mm Linux enhanced with eBPF

We compare the performance of default Linux behaviour with THP turned on and our eBPF enhanced system. The evaluation focuses on key metrics such as speedup, TLB misses using as baseline Linux with THP turned off, and the number of allocated 2MiB huge pages compared to Linux THP.

5.3.1 micro-benchmark

First we present a micro-benchmark we developed to evaluate our eBPF's system approach. The benchmark begins by allocating N memory regions of 2MiB in size (e.g. 20000 in our case). Then, we determine which percentage of these regions will be benefit most from being backed with 2MiB pages (High-Reuse TLB-Sensitive Accesses). The remaining regions are going to be sufficiently backed by 64KiB huge pages. The main part of the workload involves randomly selecting 48 regions of 2MiB, same as the number of entries of the fully associative L1 DTLB. For regions that belong to the 2MiB beneficial set we iteratively perform memory operations on the start of each 4KiB page of the region. For regions that belong to the 64KiB beneficial set memory operations are restricted in the pages of the first 64KiB of each region, ensuring that address translation overheads can be mitigated by the 64KiB huge page size.

Memory Access Pattern The following heat-map generated from the monitored access frequencies of 2MiB memory regions generated by DAMON indicates exactly the expected output. From the heat-map we observe that the orange section represents the memory access frequencies to pages included in the 2MiB huge page beneficial set. Each of the other lines, followed by a black section, represent the 64KiB huge page beneficial regions where the rest of the 2MiB region is underutilized.

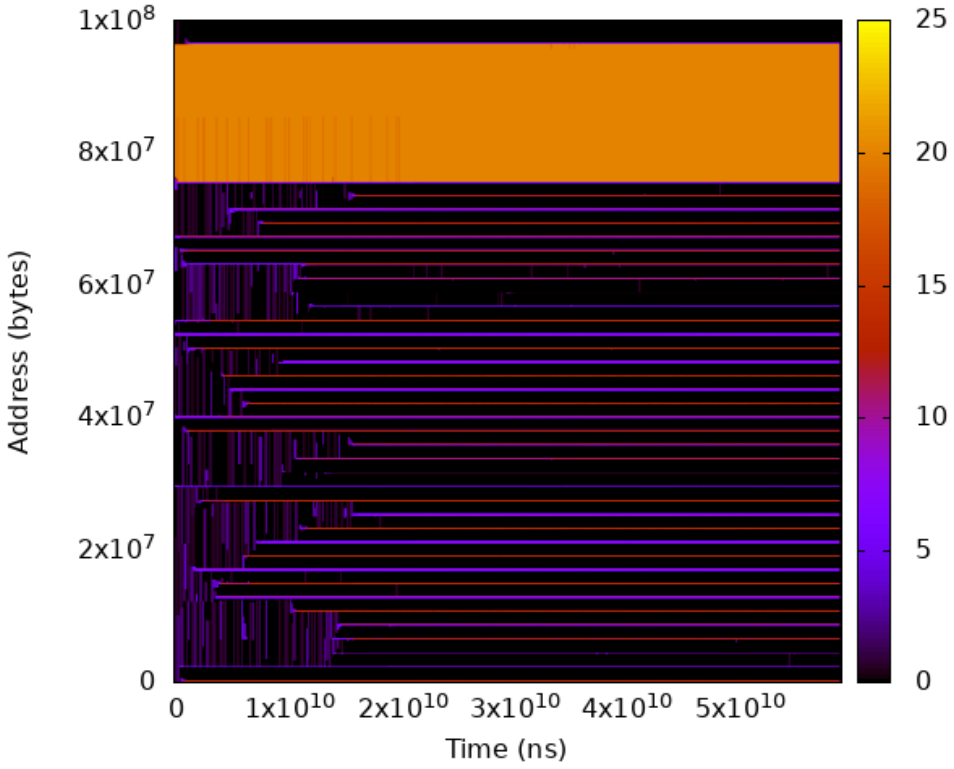


Figure 5.1: micro-benchmark memory access frequency heatmap produced by DAMON

Profile for micro-benchmark We provide the obvious profiling for the memory regions of the micro-benchmark. As previously stated, the recorded benefit is not calculated exactly, we only make sure that it surpasses the cost of zeroing the huge page, since we want to ensure promotion. We only take in mind the zeroing cost, since we run on a freshly booted Virtual Machine, with no other workload running.

```
1 0xfffff6800000,0xfffff7c00000,1,1,1,1,1,1,1,1001
2 0xfffff2000000,0xfffff6800000,1,1,100,1,1,1,1,1
```

Results As expected our eBPF-mm system is able to achieve the same performance benefits as Linux's default THP approach against the use of only 4KiB base pages. The performance increase in both scenarios stems from the fact that by using huge page, the entirety of the working set of the micro-benchmark is able to fit into the L1 DTLB. Other than, compulsory misses, both systems should not exhibit any more TLB misses regarding memory accesses to data, in contrast with using only base pages. However, our eBPF system is able to achieve the same performance increase by using 87,5% less 2MiB huge pages than Linux's THP. This way, the micro-benchmark suffers minimally from internal fragmentation and subsequently avoids producing memory bloat in the system.

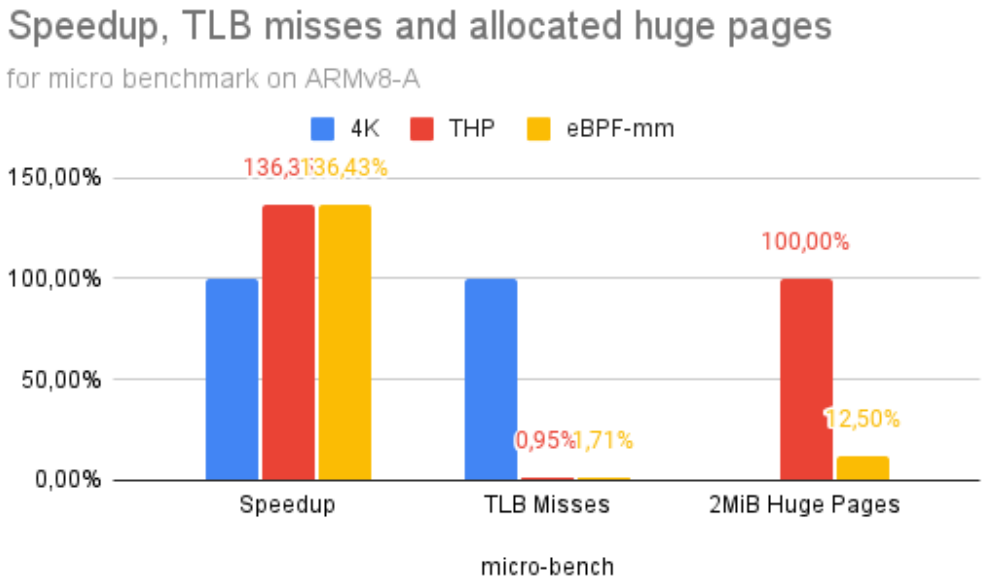


Figure 5.2: micro-benchmark results

5.3.2 astar

The astar benchmark implements the A* path finding algorithm on a 2D array representing a map. This algorithm is commonly found in computer games, artificial intelligence and path finding applications. The input of this benchmark is a map in binary format, while its output is the number of existing ways from point A to point B and the total way length needed to validate correctness.

Memory Access Pattern The heat-map generated for the astar benchmark distinguishes two memory intensive phases in the workload. In the first phase we observe that memory accesses happen frequently along the entirety of the allocated virtual address space. Some memory regions (those being closer to the orange

color) exhibit higher frequencies of memory accesses than others (those being closer to the purple color). In the second phase of the workload, the workload accesses fewer memory regions but with a higher intensity. This information alone hints that backing the entirety of the virtual address space of the astar benchmark might be the best option.

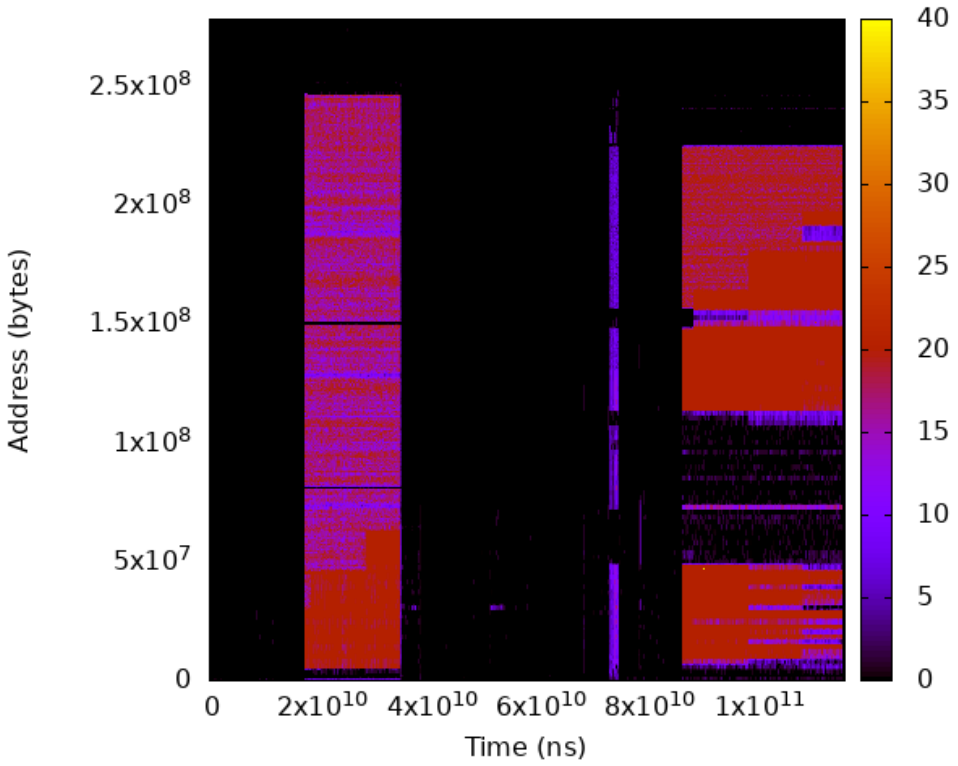


Figure 5.3: astar memory access frequency heatmap produced by DAMON

Profile for astar After studying the information provided by ARM's SPE we observe that only a fraction of the memory regions of the benchmark induce a high number of TLB misses. Following this observation, we decide to back only these memory regions with 2MiB huge pages, while in the default case a huge page of 64KiB should be allocated. This way, we aim to alleviate the translation overhead for memory region that seem to depend on 2MiB huge pages, while avoiding TLB misses in regions where a slightly larger page size than the base page could provide greater locality.

```
1 0xdc00000,0xee00000,1,1,1,1,1,1,1,1001
2 0xa000000,0xae00000,1,1,1,1,1,1,1,1001
3 0xa00000,0x1e00000,1,1,1,1,1,1,1,1001
4 0x10000000,0x10e00000,1,1,1,1,1,1,1,1001
5 0x7000000,0x7e00000,1,1,1,1,1,1,1,1001
```

```
0x9000000,0x9200000,1,1,1,1,1,1,1,1001
0xa400000,0xa600000,1,1,1,1,1,1,1,1001
0xb600000,0xba00000,1,1,1,1,1,1,1,1001
0xcc00000,0xce00000,1,1,1,1,1,1,1,1001
0x000000000000,0xffffffffffff,1,1,100,1,1,1,1,1
```

Results The results of comparing the execution of the astar benchmark on our eBPF-mm systems against the default Linux's THP aligns with the previously categorization of memory regions, regarding the expected benefit from huge pages, in real world applications. Our eBPF-mm systems results in similar performance benefits with Linux's THP, achieving similar speedup and TLB misses reduction, while also promoting only a third of Linux's THP 2MiB huge pages. This is clear evidence that in some memory regions where 4KiB pages under-perform, intermediate huge page sizes like 64KiB suffice, while 2MiB huge pages over-provision memory resources.

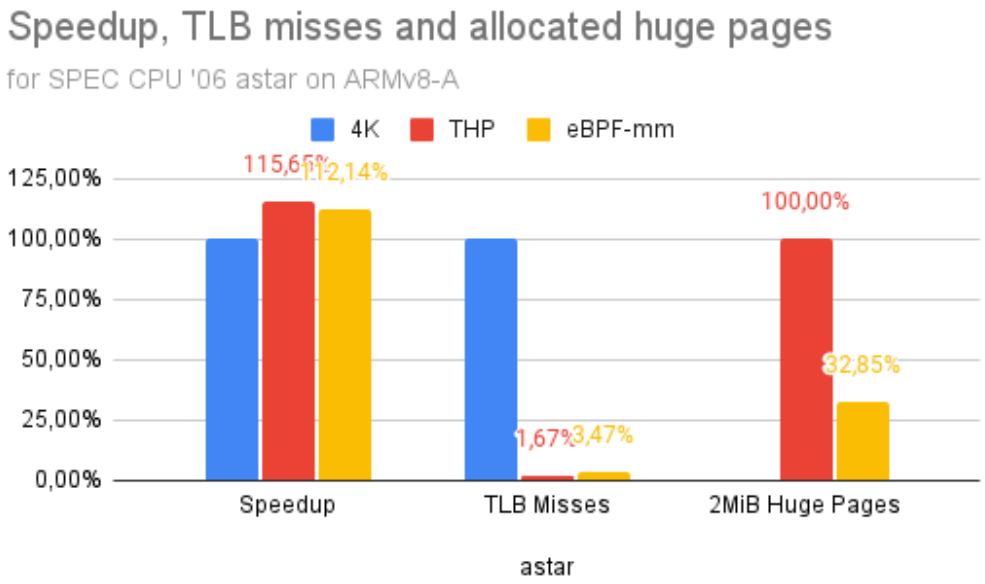


Figure 5.4: astar results

5.3.3 omnetpp

The omnetpp benchmark is a simulation of a large Ethernet network, based on the OMNeT++ discrete event simulation system, using an ethernet model which is publicly available. For the reference workload, the simulated network models a large Ethernet campus backbone, with several smaller LANs of various sizes hanging off each backbone switch. It contains about 8000 computers and 900

switches and hubs, including Gigabit Ethernet, 100Mb full duplex, 100Mb half duplex, 10Mb UTP, and 10Mb bus.

Memory Access Pattern Similar to astar, from the following heat-map generated from DAMON information we identify two, equal in duration, phases in the omnetpp workload regarding memory accesses. In the first phase, the upper half of the virtual address space exhibits a steady average access frequency along the time axis. For the bottom half of the virtual address space the frequency of memory accesses exhibit greater variability. It seems like memory regions are idle, described by pretty low access frequencies, until a certain moment in the workload where they kick start intense memory accesses. Also, the non-uniform orange coloring indicates that the intensity of these memory accesses are not steady in time, but indicate a burst pattern. In the second phase of the workload we observe very low memory accesses across the entirety of the virtual address space.

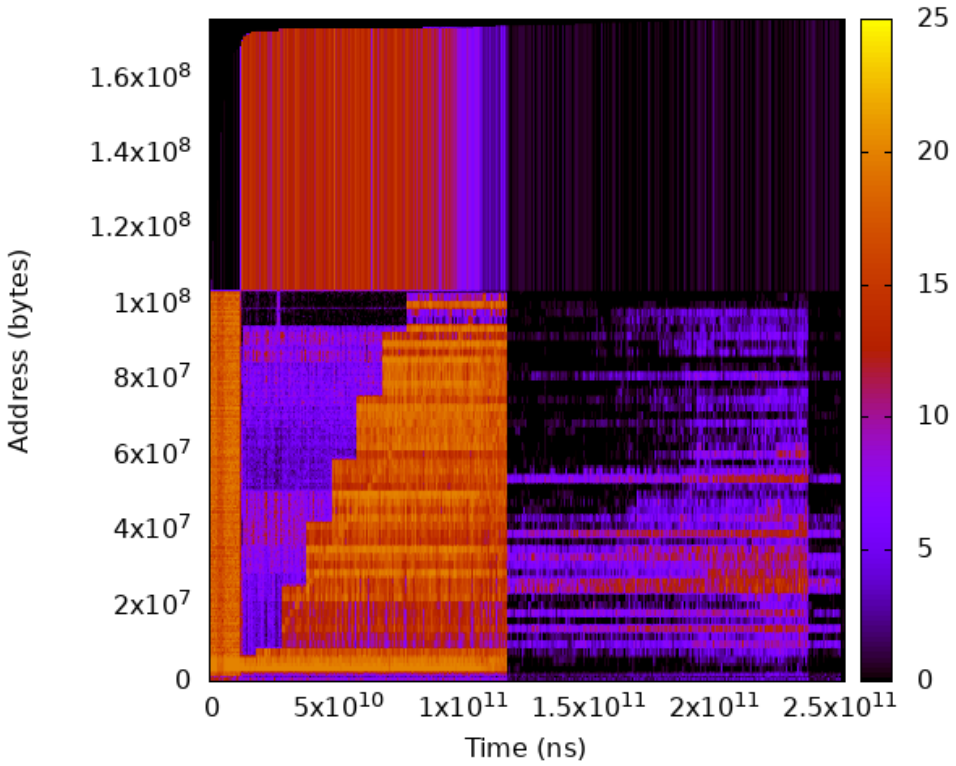


Figure 5.5: omnetpp memory access frequency heatmap produced by DAMON

Profile for omnetpp Again, combining the hints from the sampling of TLB misses with DAMON's memory access frequencies we form the following profile for the omnetpp workload. We decide to back with 2MiB huge pages the memory regions of the lower half of omnetpp's virtual address space that show the highest

amount of TLB misses. The rest of the memory pages showing average access frequencies and TLB misses we back them with 64KiB pages, while leaving some memory regions not profiled, essentially hinting the kernel to use base pages.

```
0x03000000,0x03e00000,1,1,1,1,1,1,1,1001
0x04000000,0x04e00000,1,1,1,1,1,1,1,1001
0x05000000,0x05e00000,1,1,1,1,1,1,1,1001
0x07000000,0x07e00000,1,1,1,1,1,1,1,1001
0x00600000,0x006e0000,1,1,1,1,1,1,1,1001
0x00720000,0xfffebf200000,1,1,100,1,1,1,1,1
0x4000000,0xfffebf200000,1,1,100,1,1,1,1,1
0x7400000,0xfffebf200000,1,1,100,1,1,1,1,1
0xfff633a00000,0xfffebf200000,1,1,100,1,1,1,1,1
```

Results In the omnetpp benchmark our eBPF system, although reducing TLB misses in half, could not compare against Linux's THP reducing them to 3,62% of the ones with 4KiB base pages. Apart from this fact, we observe that our system achieves similar speedup to Linux's THP while again using 42,5% less 2MiB huge pages. We attribute that to the higher induced setup latency of 2MiB huge pages while not yielding enough benefit to outperform our system. Another possibility is that further reduction of TLB misses might not yield any more significant performance benefits.

Along with the performance graphs we provide the histograms describing the latency of zeroing 2MiB huge pages in both eBPF-mm and Linux THP systems.

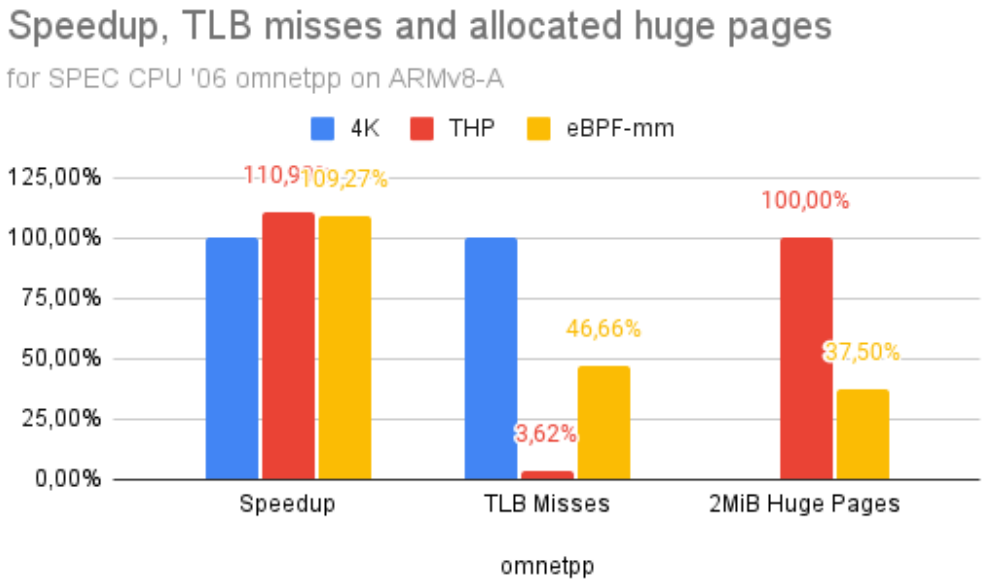
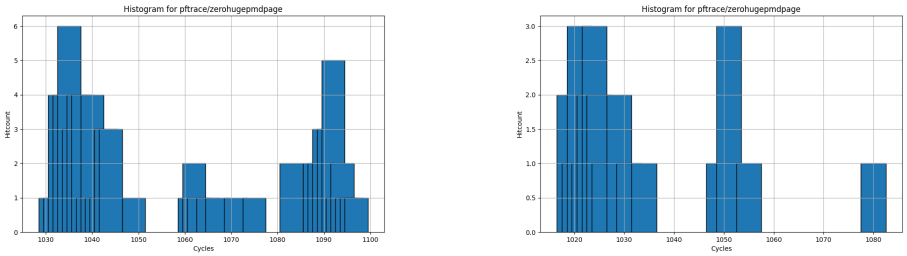


Figure 5.6: omnetpp results



(a) THP 2MiB preparation (b) eBPF 2MiB preparation

Figure 5.7: Costs of 2MiB preparation comparison

5.3.4 eBPF induced overhead

Enhancing the memory management of the Linux kernel with eBPF, certainly does not come cost-free. Each time a page fault occurs, the attached eBPF program is triggered and starts execution, looking up profiling data and calculating the optimal estimated huge page size. This adds extra overhead to a critical kernel operation. As seen in Figure **??**, the measured overhead of the execution of our eBPF program averages at 230 cycles, which is an acceptable limit.

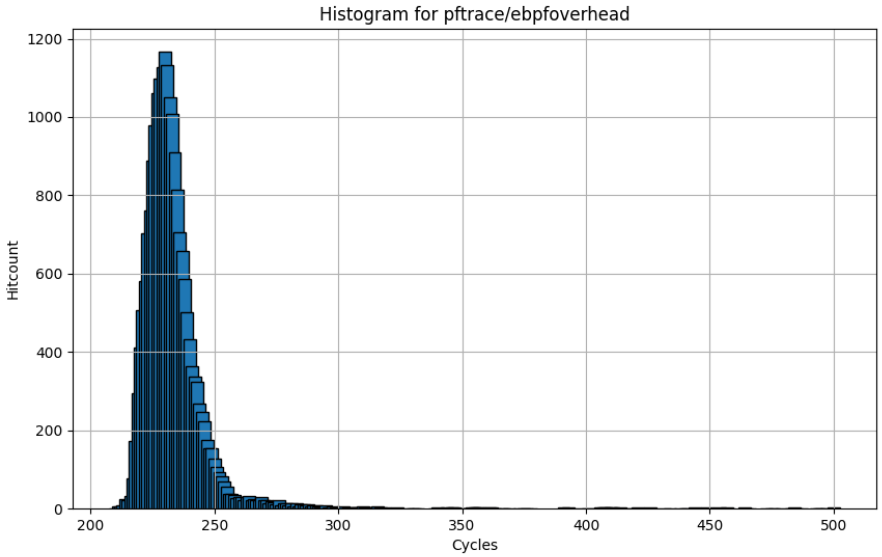


Figure 5.8: eBPF overhead

Chapter 6

Conclusions and Future Work

6.1 Conclusions

Huge pages come as a promising solution to the Address Translation overhead induced by virtual memory based memory management widespread in modern systems. Subsequently, transparent support of huge page is crucial, as it is the only approach that provides the benefits of huge pages to applications without the need to modify their source code. Huge pages do not come without trade-offs, thus operating systems need to navigate carefully the associate costs and benefits. Generalized strategies that are based on heuristics are found in the majority of modern operating systems supporting huge pages and tend to run into pitfalls.

In this work, we enhanced the Linux kernel so it can support eBPF programs that can influence decisions regarding the allocation of huge pages. Also, they are able to guide the kernel to allocate the most beneficial huge page size for a memory region, in an environment were multiple huge page sizes are available.

We use the capabilities of our system in order to implement an eBPF-enhanced memory management system based on a cost-benefit model regarding the decision between using 64KiB or 2MiB huge pages. In the workloads used for the evaluation of our system, we observe that performance similar to Linux THP can be achieved using way less 2MiB huge pages, when leveraging 64Kib huge pages in suitable memory regions. That way we are able to tackle memory bloat, while also making the programs easier to harness huge pages under fragmentation where memory contiguity might be rare. To recognize the most beneficial huge page size for each memory region, we profile the memory need of our workloads offline. Specifically, we use DAMON and ARM hardware performance counters to monitor TLB misses and the access frequency to 2MiB memory regions. Further, we observe that the overhead of eBPF programs is substantially small and does not harm overall performance.

Finally, our system makes the experimentation with synchronous huge page allocation policies easy, enabling their implementation through eBPF programs.

It also allows application developers and users to employ tailor-made, non-invasive strategies regarding huge pages, where generic policies might under-perform.

6.2 Future Work

There are several potential directions to expand upon the work presented in this thesis.

Future research could implement other huge page policies based on different criteria. For example, huge pages could be allocated based on algorithms that define and enforce fairness in huge page distribution among applications. Furthermore, our system could be enhanced to give eBPF programs the ability to decide in which memory tier to allocate a huge page in a multi-tiered memory system [19]. Another area of interest is the operating system's selection of memory regions that fall victim under reclamation algorithms under memory pressure.

In conclusion, eBPF programs could be supported to implement a wide range of policies in the memory subsystem, which depend heavily on the application's memory needs and can have a big impact on its performance.

Bibliography

[1] *Ampere® Altra® Rev A1 64-Bit Multi-Core Processor Datasheet, Rev 1.40*. https://amperecomputing.com/customer-connect/products/altra-family-device-documentation. Ampere Computing. 2023.

[2] *Arm Architecture Reference Manual for A-profile architecture, Rev. J.a.* https://developer.arm.com/documentation/ddi0487/latest/. ARM Corporation. 2023.

[3] *Arm® Neoverse™ N1 Core, Rev r4p1*. https://developer.arm.com/documentation/100616/0401/. ARM Corporation. 2023.

[4] ebpf.io authors. *eBPF Documentation*. https://ebpf.io/what-is-ebpf/.

[5] ebpf.io authors. *eBPF Documentation*. https://ebpf.io/what-is-ebpf/.

[6] Fabrice Bellard. "QEMU, a Fast and Portable Dynamic Translator". In: *Proceedings of the 2005 USENIX Annual Technical Conference*. 2005. URL: https://doi.org10.5555/1247360.1247401.

[7] Abhishek Bhattacharjee. "Preserving Virtual Memory by Mitigating the Address Translation Wall". In: *IEEE Micro* (2017). URL: https://doi.org/10.1109/MM.2017.3711640.

[8] The kernel development community. *BPF Documentation*. https://www.kernel.org/doc/html/latest/bpf/index.html.

[9] The kernel development community. *Seccomp BPF (SECure COMPuting with filters) kernel documentation*. https://www.kernel.org/doc/html/v4.19/userspace-api/seccomp_filter.html.

[10] Fabien Gaud, Baptiste Lepers, Jeremie Decouchant, Justin Funston, Alexandra Fedorova, and Vivien Quéma. "Large Pages May Be Harmful on NUMA Systems". In: *Proceedings of the 2014 USENIX Annual Technical Conference*. 2014. URL: https://doi.org/10.5555/2643634.2643659.

[11] *gperftools*. https://github.com/gperftools/gperftools.

[12] Brendan Gregg. *BPF Performance Tools*. Addison-Wesley Professional, 2019.

[13] John L. Henning. "SPEC CPU2006 Benchmark Descriptions". In: *SIGARCH Comput. Archit. News* (2006). URL: https://doi.org/10.1145/1186736.1186737.

[14] *HugeTLB Pages*. https://docs.kernel.org/arch/arm64/hugetlbpage.html.

[15] *HugeTLBpage on ARM64*. https://www.kernel.org/doc/html/latest/arm64/hugetlbpage.html.

[16] Jim Keniston, Prasanna S Panchamukhi, and Masami Hiramatsu. *Kernel Probes (Kprobes) kernel documentation*. https://docs.kernel.org/trace/kprobes.html.

[17] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. "KVM: the Linux Virtual Machine Monitor". In: *In Proceedings of the 2007 Ottawa Linux Symposium (OLS'07)*. 2007. URL: \url{https://www.kernel.org/doc/ols/2007/ols2007v1-pages-225-230.pdf}.

[18] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. "Coordinated and Efficient Huge Page Management with Ingens". In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. 2016. URL: https://doi.org/10.5555/3026877.3026931.

[19] Taehyung Lee, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. "MEMTIS: Efficient Memory Tiering with Dynamic Page Classification and Page Size Determination". In: *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*. 2023. URL: https://doi.org/10.1145/3600006.3613167.

[20] Aninda Manocha, Zi Yan, Tureci Esin, Juan Luis Aragón, Nellans David, and Margaret Martonosi. "Architectural Support for Optimizing Huge Page Selection Within the OS". In: *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 2023. URL: https://webs.um.es/jlaragon/papers/manocha_MICRO23.pdf.

[21] Mark Mansi, Bijan Tabatabai, and Michael M. Swift. "CBMM: Financial Advice for Kernel Memory Managers". In: *Proceedings of the 2022 USENIX Annual Technical Conference*. 2022. URL: https://www.usenix.org/conference/atc22/presentation/mansi.

[22] Steven McCanne and Van Jacobson. “The BSD Packet Filter: A New Architecture for User-level Packet Capture”. In: *USENIX Winter 1993 Conference (USENIX Winter 1993 Conference)*. San Diego, CA: USENIX Association, Jan. 1993. URL: https://www.usenix.org/conference/usenix-winter-1993-conference/bsd-packet-filter-new-architecture-user-level-packet.

[23] *MongoDB Docs: Disable Transparent Huge Pages (THP)*. https://www.mongodb.com/docs/manual/tutorial/transparent-huge-pages/.

[24] Juan Navarro, Sitaram Iyer, and Alan Cox. “Practical, Transparent Operating System Support for Superpages”. In: *Proceedings of the 5th ACM SIGOPS Symposium on Operating Systems Design and Implementation*. 2002. URL: https://doi.org/10.1145/844128.844138.

[25] Ashish Panwar, Sorav Bansal, and K. Gopinath. “HawkEye: Efficient Fine-grained OS Support for Huge Pages”. In: *Proceedings of the 24th ACM/IEEE International Conference on Architectural Support for Programming Languages and Operating Systems*. 2019. URL: https://doi.org/10.1145/3297858.3304064.

[26] Nikolaos-Charalampos Papadopoulos Papadopoulos, Stratos Psomadakis, Vasileios Karakostas, Nectarios Koziris, and Dionisios N. Pnevmatikatos. “Design, Implementation and Evaluation of the SVNAPOT Extension on a RISC-V Processor”. In: *CoRR* abs/2406.17802 (2024). DOI: 10.48550/ARXIV.2406.17802. arXiv: 2406.17802. URL: https://doi.org/10.48550/arXiv.2406.17802.

[27] SeongJae Park, Yunjae Lee, and Heon Y Yeom. “Profiling dynamic data access patterns with controlled overhead and quality”. In: *Proceedings of the 20th International Middleware Conference Industrial Track*. 2019, pp. 1–7.

[28] Liz Rice. *Learning eBPF*. O’Reilly Media, Inc., 2023.

[29] Ryan Roberts. *Multi-size THP for anonymous memory*. https://lwn.net/Articles/954094/.

[30] Ryan Roberts. *Transparent contiguous PTEs for User mappings”*. https://lore.kernel.org/linux-arm-kernel/87fs0xxd5g.fsf@nvdebian.thelocal/T/.

[31] *Sched$_e$xtSchedulersandTools*. https://github.com/sched-ext/scx.

[32] *The RISC-V Instruction Set Manual Volume II: Privileged Architecture*. https://wiki.riscv.org/display/HOME/RISC-V+Technical+Specifications. RISC-V Foundation. 2021.

[33] *Transparent Hugepage Support*. `https://www.kernel.org/doc/Documentation/vm/transhuge.txt`.

[34] Wm A Wulf and Sally A McKee. "Hitting the memory wall: Implications of the obvious". In: *ACM SIGARCH computer architecture news* 23.1 (1995).